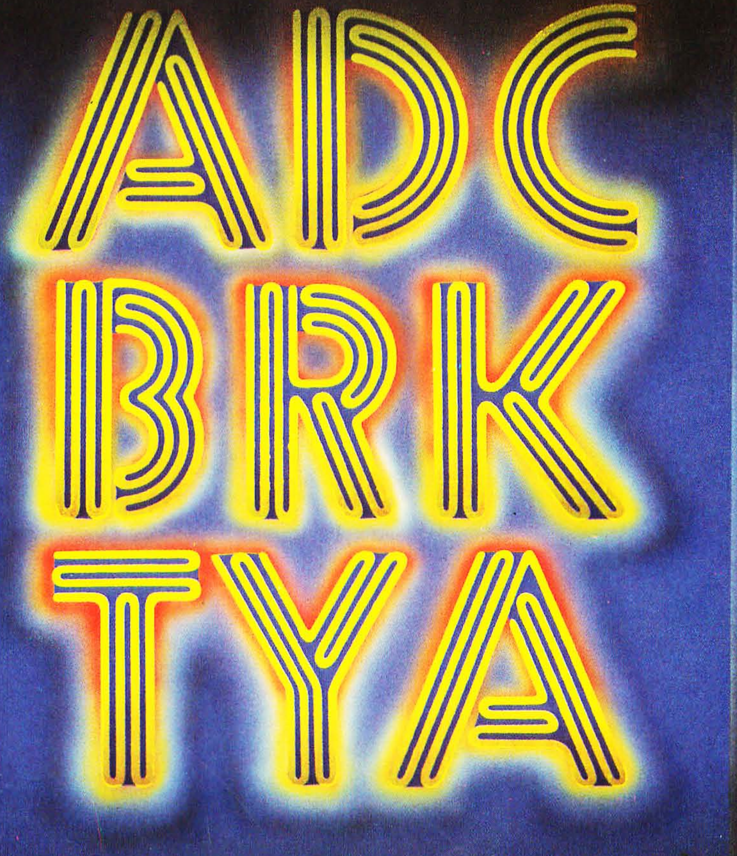
A BYTE BOOK

>$15.95

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

MARK CHASIN



ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

SOFTWARE AVAILABLE

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

MARK CHASIN

McGraw-Hill Book Company

New York St. Louis San Francisco Auckland
Bogotá Guatemala Hamburg Johannesburg
Lisbon London Madrid Mexico Montreal
New Delhi Panama Paris San Juan
São Paulo Singapore Sydney Tokyo Toronto

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

A BYTE Book.

1 2 3 4 5 6 7 8 9 0 SEM SEM 8 9 3 2 1 0 9 8 7 6 5 4

ISBN 0-07-010679-7

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA
Chasin, Mark.
Assembly language programming for the Atari computers.
(A Byte book)
Includes index.
1. Atari computer — Programming. 2. Assembler language (Computer program language) I. Title. II. Series.
QA76.8.A82C43 1984 001.64'.24 84-11214
ISBN 0-07-010679-7

The editor for this book was Barbara Brooks; and the editing supervisor was Marthe Grice.
Book design by Sharkey Design.

CONTENTS

PREFACE

Since you've picked this book up and started browsing through it, you probably own or have access to an ATARI computer and are interested in progressing beyond BASIC. As you already know, the ATARI computers are among the most impressive of all home computers, but many of their special features are not available from BASIC.

This book is designed to teach assembly language programming to anyone who understands ATARI BASIC. Yes, anyone! You've probably read other books and articles which create a mystical aura around assembly language, or else use the phrase **machine language** as if it were the secret code to unlock the door to the Thief of Bagdad's treasure troves. Of course, only the privileged get to look at this secret treasure.

Bunk! Anyone who has programmed in BASIC, or any other language for that matter, can learn to program in assembly language, given the desire and the correct instruction in the language. This book provides the tools you need. Each programming language — BASIC or PILOT or FORTH or, yes, even assembly language — has its own words which stand for certain operations. One example is PRINT in BASIC, which directs information to your TV screen. The combination of these words, and the way they must be strung together to make the computer do what you want it to do, is called the **syntax** of the language.

In this book, you will learn the syntax of assembly language, and you will also learn, by frequent examples, how to use assembly language to make your ATARI perform tasks which are either impossible from BASIC or 200 times slower in BASIC. The examples are fully documented both by frequent remarks and by a thorough discussion of the purpose, programming techniques, and theory, where appropriate, of each program. This discussion allows you to progress beyond the examples and to write your own subroutines or even whole assembly language programs for the ATARI. Furthermore, the routines in this book follow the "rules" established by ATARI for assembly language programmers, so they will work with any ATARI computer, from the earliest 400, to the most advanced 1450XLD, and everything in between.

Examples are given both in assembly language and, wherever possible, also in BASIC programs which incorporate these assembly language routines to perform tasks from BASIC. These routines can be used immediately in your own programs. In fact, you can use the enclosed order form to obtain on disk all assembly language and BASIC programs in this book. The disk is ready to run or modify for your own uses. Included on disk and here are such techniques as reading the joysticks, moving players and missiles, input or output to all possible devices such as printers, disk drives, cassette recorders, the screen and more, vertical blank interrupt routines, display list interrupts, fine horizontal and vertical scrolling, sound, graphics — in short, everything you've always heard the ATARI computers were capable of but had no idea how to program.

One entire chapter of the book is devoted to the use of assemblers and how to use this book with any of the many fine assemblers available for the ATARI computers. You'll need an assembler, just like you need BASIC to program in BASIC, and this book will interact with any of them.

If you've reached the point where BASIC is no longer enough, and you'd like to progress to a language which gives you absolute control over all functions of your remarkable computer, then begin with Chapter 1, and you'll see how easy it is. Who knows, maybe you'll be the one to write the sequel to STAR RAIDERS!

Mark Chasin

PART ONE
BACKGROUND

CHAPTER ONE
INTRODUCTION

Welcome to the world of assembly language programming for the ATARI computers. By now, you've no doubt tried your hand at programming your ATARI in BASIC and found it to be a very easy-to-use and powerful language. But you've also probably found some things that just can't be done in BASIC, and you know that all of the excellent real-time action games and the fast sorts and searches are all programmed in some mysterious language called **machine language**. The purpose of this book is to teach you how to program your ATARI in the fastest, most powerful and versatile language available, assembly language. By working your way through this book, you will learn how to use all of the sophisticated and powerful resources of one of the most impressive home computers, the ATARI.

Most of the examples in this book will be related to BASIC, so an understanding of BASIC will be important to the understanding of this book. However, many types of programs that can be written in assembly language simply have no counterparts in BASIC, and so for these no such examples will be possible. Problems will be presented throughout the book and it is highly recommended that you try to work them out for yourself. In each case the answers will be presented and discussed, in order to help you if you are having trouble.

VARIETIES OF PROGRAMMING LANGUAGES

At a very fundamental level, your ATARI really only understands one programming language, which is called machine language, the language of the computing machine. A typical machine language program might look like this:

```
1011010110100101.....
```

Now, before you put this book down and go back to BASIC, let's understand one thing right away: virtually no one programs directly in machine language. Even the many programs advertised as being written in "100% machine language" weren't; they were written in assembly language and then translated into machine language. But all computer languages must at some time be translated into machine language in order to be executed, even BASIC. That's right, the central "brain" of your ATARI computer doesn't even really understand BASIC.

BASIC: AN INTERPRETED LANGUAGE

Let's spend a moment discussing how a BASIC program is executed, in an effort to understand better what assembly and machine language really are, and how they differ.

Let's first write a very simple BASIC program:

```
10 PRINT "HELLO"
20 FOR I=1 TO 200
30 NEXT I
40 PRINT "GOODBYE"
50 END
```

If we now type RUN and hit the RETURN key, we know that the word HELLO will appear on our TV or monitor screen and, after a brief pause, the word GOODBYE will appear directly below it, followed several lines later by the word READY. But exactly how does this happen?

The cartridge containing ATARI BASIC is actually more properly called the ATARI BASIC Interpreter. An interpreter, just like the noncomputer use of the word, is someone or something that translates information from one form into another, whether from English into Russian, or from BASIC into some other language. In our case, the BASIC cartridge contains a program that can translate BASIC keywords into a form understandable to our computer's "brain." Let's see how.

As we type line 10, the word PRINT is translated to a code for the word PRINT, called a **token.** This process is called tokenizing your BASIC program, and is done as you type each line into your ATARI, and hit RETURN. It is this process that simultaneously checks the syntax, or grammar rules, to be sure that you typed the line correctly. If not, you'll see the familiar ERROR statement immediately after typing the line, and you then can correct your mistakes before proceeding. This ensures that when the BASIC cartridge begins interpreting your program, it may have logical errors to deal with, but at least each line is internally correct.

Having completely typed the above program, we would then type RUN and press RETURN, which would begin the interpretation of the program. The first thing this interpreter knows is that the beginning of the program, the place it must start when the word RUN is typed, is the lowest-numbered line of the BASIC program. Actually, before it ever gets there, it does quite a bit of housekeeping, such as setting all variables used in your program to zero, canceling out any previously used strings or arrays, and many other functions. Then it turns its attention to line 10, which is converted into machine language by means of something called a jump table, about which we'll learn a great deal in Chapter 9. In any case, first line 10 is translated, then it is executed, and then the machine language code is thrown away, to make room for the next line, line 20. The process of translation, execution, and discarding is repeated for line 20 and then again for line 30, and so on.

Having now executed the entire program, and seen the READY prompt that tells us that BASIC is ready for new instructions, what do you suppose will happen if we type RUN again? Right! The entire process of translating, executing, and discarding each line will be repeated all over again. Then we'll see the READY prompt again. In fact, this entire process will occur as many times as we choose to type the word RUN. As you can no doubt see, this is a very wasteful process. BASIC continues to repeat over and over two of the three steps which are not actually needed to run the program, translation and the discarding of information. If we could only get away from the need for these two steps, imagine how fast our program would execute. After all, if we get rid of these two steps, the only one left is execution.

ASSEMBLY LANGUAGE: AN ASSEMBLED LANGUAGE

Now you know the purpose of assembly language programming! When we program in assembly language, by using a translator known as an **assembler**, we can produce the executable machine language code which we can store, and which the computer can execute directly. We translate it only once and we don't discard it at all, so we get maximum efficiency, and therefore, maximum speed. And that's the real benefit of assembly language programming, speed. In fact, it is possible to write a program in assembly language which will execute over 1000 times faster than its BASIC equivalent! For arcade games, and very time-consuming processes like moving blocks of memory around, searches, sorts and other such procedures, assembly language programming can be absolutely indispensable.

The other major advantage of assembly language is the absolute control it gives the programmer over the computer. In BASIC, the programmer is often separated from the nuts-and-bolts hardware of the computer and doesn't have detailed control over many of its functions. This control is available only through assembly language programming.

INTERPRETED VERSUS ASSEMBLED LANGUAGES

These are the advantages of assembly language programming: speed and control. How about the disadvantages? First, of course, is the need to learn a new computer language. This book will enable you to do that. Second, ATARI BASIC is an interpreted language, while assembly language is not. This becomes important when you need to make changes in a program. In BASIC, you simply make the change and rerun the program. For example, to change the above program, we might simply type:

```
40 PRINT "GOODBYE";
50 PRINT "Y'ALL"
60 END
```

Now when we run the program, it will say GOODBYE Y'ALL instead of just GOODBYE, as above. The entire change in the program might take 15 seconds for a very slow typist. This flexibility is a great advantage of interpreted languages. To make a similar change in an assembly language program would require much more typing, and then the program would have to be reassembled. This assembly process, converting the assembly language program to machine language, sometimes takes 15 minutes or more, depending on the size of the program and the assembler used. Of course, our example is very short and would not take this much time, but the point is that making even a very simple change to an assembly language program might take quite a while, and if you make a mistake, you'll need to repeat the process all over again!

A third disadvantage of assembly language is the amount of programming you'll need to do to accomplish even the simplest tasks. For instance, the PRINT statement in BASIC, which requires you to type only one word, might require 20 or 30 **lines** of programming in assembly language. For this reason, assembly language programs are usually very long.

The fourth, and last, disadvantage of assembly language is the difficulty of understanding a printout of the program. Certainly the PRINT statement in BASIC is far more understandable than a series of instructions such as:

```
LDA #$01
STA CRSINH
```

or something equally obtuse. This problem can and should be overcome by all good assembly language programmers by the inclusion of comments on virtually every line. Comments are the assembly language equivalent of REM statements in BASIC: they help the programmer to remember what it was he or she was trying to accomplish with a given line. Certainly the above example makes somewhat more sense when presented below with comments, even to someone who doesn't understand assembly language at all.

```
LDA #$01       ;to inhibit cursor
STA CRSINH     ;poke a 1 here
```

Now perhaps it's more understandable that when we see a program advertised as written in "100% machine language," what is really meant is that it was written in assembly language, and then translated once from its final form into machine language, which is the form in which it is being sold. Such programs generally are much faster to execute than BASIC programs, and the additional control the programmer has over the computer allows special effects not attainable from BASIC.

There is an additional distinction between BASIC and assembly language. BASIC belongs to a family of programming languages which are referred to as **high-level** languages. This nomenclature refers to the ability of one simple statement to perform quite a complicated task, such as the PRINT example used above. In a sense, this ease of programming also isolates the programmer from the hardware, placing him or her at arm's length, so to speak. It is from this view of languages such as BASIC that the term **high-level language** arose. Among *thousands* of other high-level languages are Pascal, FORTRAN, PILOT and Ada. In contrast to these, languages such as machine language or assembly language are referred to as **low-level** languages, because to program using them requires an understanding of the hardware and an ability to get into the real guts of the machine for which you are programming.

WORKING WITH ASSEMBLY LANGUAGE

In order to convert an assembly language program to machine language, we must use another program, called an **assembler**. There are a number of excellent assemblers available for the ATARI computers, and the techniques used in this book will work with any of them. Chapter 6 is devoted to the syntax and special functions of each assembler, but the assembly language programs listed in this book were produced using the Assembler/Editor cartridge from ATARI. Chapter 6 specifies all of the changes required to use these programs with each of the other assemblers.

COMPILERS

There is another way to convert programs to machine language. A compiler is a program which converts a program written in a high level language such as BASIC to machine language. These compilers generally convert the entire program all at once, in contrast to an interpreter, which translates each line one at a time. The converted program created by the compiler can then be run without a BASIC cartridge installed, and will generally be from five to ten times faster than the original BASIC program. Why only five to ten times? These compilers are very complex programs, which must take into account every possible combination of BASIC commands anyone might write. Therefore, they create machine language code which performs all of the correct steps in the original program, but they cannot optimize the code produced. Therefore, in general, programs written in assembly language and assembled into machine language will execute much faster than the same program written in BASIC and compiled.

The other major disadvantage of compiled code is its size. For instance, some of the subroutines in Chapter 7 are about 100 bytes long. The same routines written in BASIC and compiled could be as long as 8000 bytes! It would be very hard to use these as subroutines in a BASIC program as we do in Chapter 7.

TERMINOLOGY

Before we go on, let's talk about a number of terms that are frequently used by programmers. It's the jargon of their trade. Just so we all are speaking the same language, then, let's briefly review some of them. When we speak about computer memory, we frequently hear the terms ROM and RAM mentioned. ROM stands for **R**ead-**O**nly **M**emory, and memory of this type can be read but not written to. For instance, in the ATARI, all memory locations higher than 49152 are ROM, and although in BASIC we can PEEK them to see what is stored there, we cannot POKE new values into them. "But what about player-missile graphics?" you may ask. "We POKE memory locations higher than this all the time!"

True, but if you were to then PEEK at that location, you would find that you hadn't really changed anything at all. The value stored in that location is not changed by such POKEs. It is the act of writing to that address which causes the changes you see in player-missile graphics or other applications requiring writing to memory locations above 49152.

This is in direct contrast to RAM, which stands for **R**andom-**A**ccess **M**emory. Actually, both ROM and RAM are random-access, and RAM should more properly be called Read-Write Memory; but since RWM is unpronounceable, RAM has become the accepted term. The term **random access** refers to the method by which information is accessed, and is to be contrasted with **sequential access**, the other major method of storage. Sequential access can best be envisioned by imagining an audio tape. In order to play a song in the middle of the tape, you must somehow scan through the entire first portion of the tape, either by playing it or by using the fast-forward key. In contrast, think of a phonograph record. To play the middle song on a side, we simply lift up the tone arm and bring it down on the song we want, which immediately begins playing. We have not had to go through any other songs to get to that one. The audio tape is a sequential-access device, as is a computer tape, such as the ATARI 410 program recorder, and the phonograph record is a random-access device, as is a computer disk, such as that used in the ATARI 810 disk drive.

The next terms, with which you may or may not be familiar, are OS and DOS. OS stands for **O**perating **S**ystem, and your ATARI has one of the best operating systems of any microcomputer. The operating system is contained in ROM (Remember? Read-Only Memory!) in your computer, and is responsible for controlling almost everything that happens inside your ATARI. Without the operating system, nothing would happen when you turned on your computer. The operating system has complete control over every facet of computing. We'll learn how to interact with this fine operating system in considerable detail as we work our way through this book.

Perhaps it should be said that the ATARI has several of the best operating systems of the popular microcomputers, since the operating system for the 400 and 800 is slightly different from that of the 1200XL, which in turn is slightly different from that of the 600XL, the 800XL, and the 1450XLD. In fact, even the 400 and 800 had two different versions of their operating systems, the so-called A and B ROMs! How are we to begin programming for so many different operating systems?

This is the nicest part about the operating system for the ATARI computers. ATARI has guaranteed that certain vectors in the operating system will never change. A **vector** is a signpost, a directional indicator. It tells us how to find particular routines or where to find a certain part of the operating system. With this information, it is possible to write a program which will work not only on our 400 or 800 or on an 800XL, but even on generations of ATARI computers which ATARI themselves have not yet dreamed of producing!

There are a number of shortcuts around these vectors available in the ATARI computers, but there is no guarantee that programs which use these shortcuts will work on all ATARIs. For this reason, they are strongly *not* recommended for general use. Of course, if you're just writing a quick and dirty subroutine for your own program, to use on only your computer, these shortcuts are useful, but many programs written in assembly language have failed as soon as new operating systems were made available by ATARI. In one case, such lack of foresight has even caused the untimely demise of a third-party software house, so if you're contemplating selling what you write, *be sure to obey the rules.*

The related term, DOS, stands for **D**isk **O**perating **S**ystem. This is the program that controls any disk drives which may be connected to your ATARI. It actually consists of two parts, DOS.-SYS and DUP.SYS. The DOS.SYS portion of DOS is loaded into your computer when you first turn it on, and is always present. The DUP.SYS portion of DOS is only loaded when you type DOS from the keyboard. It contains the familiar DOS menu allowing many of the usual file manipulation commands, such as copying disks, saving areas of memory, formatting disks, and many others. You should note that there are no guaranteed vectors in DOS, although so much software depends on certain locations that changes in these would have to be considered unlikely. But you never can tell.

Now that you know the difference between languages, and between interpreters, assemblers, and compilers, we'll next explore the various numbering systems used by our computers.

CHAPTER TWO
GETTING STARTED

NUMBERING SYSTEMS IN GENERAL

Before we can learn assembly language programming, we must first review several different numbering systems used in such programming. Let's first review the decimal system, the one with which we are most familiar. The decimal system is based on ten, most likely because we have ten fingers, and counting using our fingers was the simplest form of arithmetic for early peoples.

Think about the number 123 for a moment. Exactly what does this number represent? If we think about what we learned in school, we remember 1 one hundred, 2 tens, and 3 ones, which, when added together, total 123. There is, however, another way of looking at this number. It turns out that each digit in the decimal system (base 10) is 1 power of 10 higher than the digit to its immediate right. For those of you who don't clearly remember what a power is, that term simply tells you how many times the base is multiplied by itself. For instance, 10 to the power of 3 is $10 \times 10 \times 10 = 1000$, or 10 multiplied by itself 3 times.

Now, to return to 123, we remember that in any numbering system, the right-hand-most digit is always the ones column. Why is this? Because that digit is always the base — in this case, 10 — to the zero power. Anything to the zero power is always 1, so this digit is always the ones digit. In our example we get $3 \times 1 = 3$. The next digit, the 2 in this case, represents the base 10 to the first power, or 10. Since 2 × 10 = 20, we get the correct middle digit for our number. Finally, the left-most digit, 1, represents the base 10 to the second power, or 100. Therefore, this digit represents 100 × 1, or 100, and the whole number represents 100 + 20 + 3, or 123. If we view the number schematically, this process becomes somewhat easier to follow. In the examples below, we always start from the right, and progressively move toward the left. For instance, we can represent the decimal number 123 as:

The base	To the power of	Equals	Times the digit	Equals the value
10	0	1	3	3
10	1	10	2	20
10	2	100	1	100
			Total =	123

Let's pick a slightly more complicated number and go through it one more time, using 53,798.

The base	To the power of	Equals	Times the digit	Equals the value
10	0	1	8	8
10	1	10	9	90
10	2	100	7	700
10	3	1,000	3	3,000
10	4	10,000	5	50,000
			Total =	53,798

THE BINARY NUMBERING SYSTEM

Now that we've seen how to take apart a decimal number, the kind with which we're all so familiar, let's move on to a different numbering system, the binary system. Why binary? Computers really are relatively simple devices, and the fundamental piece of information stored in them is called a **bit**, or **b**inary dig**it**. A bit is the smallest amount of information; it can either be on or off, yes or no, a 1 or a zero. In a sense, a bit is like a standard light switch.

Excluding dimmers for a moment, a light can either be on or off; there is no in-between. A bit in a computer behaves exactly the same way. This explains why the binary numbering system is such a natural system for computers. The binary system consists of only two digits, zero and 1. Therefore, anything represented in the binary system can immediately be understood by our computer, which really can understand only these two digits.

Learning a new numbering system could be a real chore, but we'll make it easy. In fact, all numbering systems work exactly the same way. The only difference between them is the base used. The decimal system, as we have seen, uses 10 as the base. The binary system uses 2 as the base. Note that the largest digit in any numbering system is always 1 less than the base. For example, 9 is the largest digit in base 10, and 1 is the largest digit in base 2. Why? Because we have to allow for the digit zero, and in every numbering system the total number of different digits is equal to the base for that numbering system.

To show you how easy it is to understand the binary system, we'll take it apart just as we did the decimal system above, using the binary number 10110110.

The base	To the power of	Equals	Times the digit	Equals the value
2	0	1	0	0
2	1	2	1	2
2	2	4	1	4
2	3	8	0	0
2	4	16	1	16
2	5	32	1	32
2	6	64	0	0
2	7	128	1	128
			Total =	182

Therefore, 10110110 in the binary numbering system is equal to 182 in the decimal numbering system. Let's try one more, only this time you try it by yourself first, before you look at the answer, and then we'll work it out together below. The binary number to convert is 01011101.

The base	To the power of	Equals	Times the digit	Equals the value
2	0	1	1	1
2	1	2	0	0
2	2	4	1	4
2	3	8	1	8
2	4	16	1	16
2	5	32	0	0
2	6	64	1	64
2	7	128	0	0
			Total =	93

Did you get it by yourself? The binary number 01011101 is the same as the decimal number 93.

Now we know how to get from a binary number to a decimal number, but how do we reverse the process, in order to get from a decimal number to a binary number? That's even easier. Just take your decimal number and successively divide by the powers of 2, starting with the highest power which will divide into your decimal number with a result of 1. Each time, divide the remainder by the next-lower power of 2. For instance, let's convert 124 to the binary system.

124/128 = 0	with a remainder of 124
124/64 = 1	with a remainder of 60
60/32 = 1	with a remainder of 28
28/16 = 1	with a remainder of 12
12/8 = 1	with a remainder of 4
4/4 = 1	with a remainder of 0
0/2 = 0	with a remainder of 0
0/1 = 0	with a remainder of 0

The number 124 in the decimal system equals 01111100 in the binary system.

THE HEXADECIMAL NUMBERING SYSTEM

Now we're almost finished with numbering systems. One more step to go, and we will be finished. There is a much easier way to represent numbers than the binary system. This system is called the

hexadecimal system. Hexadecimal? Hexadecimal stands for 16, and the base of the hexadecimal system is, not suprisingly, 16. We already know that the largest single digit in this system must be 15. Wait a minute! The number 15 is not a single digit. So we need some new way of representing the numbers from 10 to 15. The easiest symbols to remember are the first six letters of the alphabet. Therefore, in the hexadecimal system, the number 10 is represented by the letter A, 11 by B, and so on, up to 15, which is represented by the letter F. Since the base of the hexadecimal system is 16, the values of the digits increase by powers of 16. An example is in order. Let's translate the number 6FC in hexadecimal nomenclature into the decimal system.

The base	To the power of	Equals	Times the digit	Equals the value
16	0	1	C	12
16	1	16	F	240
16	2	256	6	1536
			Total =	1788

So hexadecimal 6FC represents decimal 1788. In most computer articles and texts, hexadecimal numbers are preceded by a dollar sign, so the proper way to represent the decimal number 1788 in hexadecimal nomenclature is \$6FC. To convert from decimal to hexadecimal, divide as was shown above, but instead of dividing by powers of 2, divide successively by powers of 16:

```
1788/256 = 6        with a remainder of 252
  252/16 = 15  (F)  with a remainder of 12
    12/1 = 12  (C)  with a remainder of 0
```

One more conversion to go, and this is by far the easiest of all. Let's use the binary number 10110111. We already know that this is equal to the decimal number 183, and we could convert this to its hexadecimal equivalent. But this is the long way around. We will frequently need to convert from binary to hexadecimal, so let's learn how to do it directly, in one very easy step.

We first take the binary number, 10110111 in this case, and break it into two parts, right down the middle. If the 8 bits are called a byte, then each set of 4 bits should be called . . . a **nibble**. And it is! The high-order nibble is 1011, and the low-order nibble is 0111. Each of these nibbles can easily be converted to a single hexadecimal digit, since four bits represents a number from zero to 15.

1011 =	1 eight	= 8		0111 =	0 eights	= 0
	0 fours	= 0			1 four	= 4
	1 two	= 2			1 two	= 2
	1 one	= 1			1 one	= 1
	Total	= 11 (B)			Total	= 7

Thus, the hexadecimal equivalent of 10110111 is $B7, obtained directly without going through a decimal number as an intermediate (Fig. 2-1). You can work out for yourself that the number is the same no matter how you obtain it.

Byte = 1011 0111 binary

↓ ↓

High order nibble	Low order nibble
1011 binary	0111 binary
or	or
$B hexidecimal	$7 hexidecimal

↓ ↓

Byte = $B7 hexidecimal

Fig. 2-1

To translate from hexadecimal into binary, just reverse the above process. Here's how the hexadecimal number $FA is represented in binary nomenclature:

F = 1 eight	C = 1 eight
1 four	1 four
1 two	0 twos
1 one	1 one
1111	1101

11111101

ORGANIZATION OF DATA

Now that you can easily convert numbers from one base to another, let's talk for a moment about how data is organized in your computer. You'll have noted already that in all of the above examples, the binary numbers were organized into groups of eight digits. In computer jargon, 8 bits form a **byte**. Each memory location in your ATARI stores 1 byte of information. It should be apparent that in the largest possible byte, all of the bits are equal to 1. You can now calculate from this that the largest single byte any computer can store is decimal 255. By the same logic, there are only 256 possible different bytes (remember zero). So how does the computer handle larger numbers, and how does it handle more than 256 different numbers?

Computers can handle larger numbers in two different ways. One is to couple several bytes together to represent a single number. Using this technique, a 2-byte number can be as large as 256 × 256, or 65,536. Although 3-byte numbers are not normally used in your ATARI, this system allows numbers as large as 256 × 256 × 256, or 16,777,213. As you can see, this technique will allow storage of very large numbers. The second method of large number storage is to use floating point numbers. The numbers used so far in this chapter have all been integers; that is, they have all been whole numbers. No fractions or decimals can appear in an integer. However, there are no such restrictions on floating point numbers. Numbers such as 1.237 or 153.2 are perfectly valid floating point numbers, whereas they are not valid integers. The term **floating point** comes from the concept that the decimal point can float from place to place, as in the two decimal floating point numbers just described. In both cases, there were four digits in the numbers, but in the first case, three were to the right of the decimal point, and in the second, only one was. How do we represent such numbers in a computer?

In general, the numbers are coded so that 1 byte represents the power of 10 by which the number is multiplied, 1 byte represents the sign of this power (whether the number is greater than one, or between zero and one) and several bytes represent the mantissa, or the number itself. In other words, we could code 153.2 as the following sequence of bytes:

```
1,2,1,5,3,2
```

In the coding scheme used here, the first digit represents the sign of the exponent, with 1 being positive and zero being negative. The second digit represents the power of 10 by which to multiply the mantissa, or 100. The rest of the digits represent the number itself, with the decimal point understood to be after the first digit. Therefore, decoding this number according to these rules gives:

$$100 \times 1.532 = 153.2$$

Of course, many other coding schemes are possible, but the main idea is that by coding numbers, very large numbers can be represented in a computer, even using no byte greater than 255.

In our discussion so far, we have concentrated entirely on positive numbers. How do we handle negative numbers? By using signed binary arithmetic. In this system, the left-hand-most bit, called the **most significant bit**, doesn't represent a power of 2 at all, but rather represents the sign of the number. That is, if the most signficiant bit is 1, the number is negative, and if the most significant bit is zero, the number is positive (Fig. 2-2). One fallout of this system is that the largest signed number we can represent in 1 byte is +128, or −127, since we only have 7 arithmetic bits with which to work.

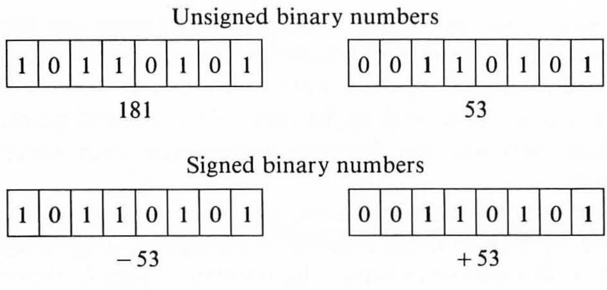


Fig. 2-2

One note of caution is warranted here. If you are using 2-byte signed arithmetic (and the ATARI does this frequently) only the most significant bit of the most significant byte (the byte representing the highest digits of the number) is the sign bit. That is, a 2-byte signed number contains 15 numeric bits and only 1 sign bit. This will become very important later when we get into 2-byte math. Of course, when we use floating point arithmetic, we need to add to our code 1 byte which will represent the sign of the number — that is, whether the final number represented is positive or negative, but all coding schemes used do take this into account.

MEMORY ADDRESSING TECHNIQUES

In BASIC, making reference to a specific memory location is fairly simple and straightforward. For instance,

```
POKE 752,1
```

is a direct command to place the value 1 into memory location 752. Additionally, we know that the maximum random-access memory available in a standard ATARI computer is 48K RAM. With the 10K ROM operating system, and other space taken for other specific purposes, the maximum total memory allowable in a normal ATARI is 64K, or 65,536 memory locations. This number should sound somewhat familiar, since we have encountered it before. It is the largest number which can be coded by 2 bytes.

The ATARI addresses memory by using a 2-byte system, which allows it to address 65,536 different memory locations. Every computer based on the 6502 chip has the same built-in limitation on the number of different memory locations which can be addressed. So, how can some ATARIs contain more than this amount of memory? How can some 6502 computers boast of more than 64K of total memory, ROM and RAM combined?

The secret to this increased memory addressing is to use a technique called **bank selecting** memory. Using this procedure, another byte is used to control which bank of memory the 2-byte addressor will reference. Imagine a computer with 16 banks, or tiers, each of which contains 64K of total memory. Only 1 of these 16 banks could be used at any one time, but all might be available from time to time. If the bank-selective byte is equal to 0, the first bank, which contains the normal 48K of RAM and the ATARI operating system, is selected. Under these conditions, if the addressor says that it wants information from location 752, it retrieves information from the normal location 752, just like an unmodified ATARI. If, however, the bank-selective byte is equal to 1, the first 64K bank of RAM is selected. Another PEEK at location 752 would now choose a location in this bank of memory, which would probably contain information totally different from that contained in the previous example.

As you can probably imagine, some aspects of bank-selective memory are particularly tricky to handle. For instance, imagine running a BASIC program stored in the normal 48K. Halfway through the program, we access a new bank of memory. All of a sudden, the computer loses track of the BASIC program it was running, since the program is no longer present in the addressable space of the computer, at least until we reselect the appropriate bank. This would result in a crash, and we would probably lose both our program and the information we were trying to access. These problems can be overcome to some extent, and third-party software and hardware vendors already have products on the market which will allow expansion of your ATARI beyond the usual maximum of 48K RAM. There is no theoretical reason why you cannot have an ATARI in your home with a maximum addressable memory of over 16 million bytes (yes – million!). In fact, don't be suprised if sometime soon you see expansion systems available for the ATARI that will take it well beyond the 192K maximum currently available. Such products, along with mass storage devices currently under development, will allow your ATARI to run programs as yet undreamed of by even the most diehard ATARI user. To take advantage of these systems, a thorough understanding of assembly language programming and of the construction of your ATARI will be necessary, and the remainder of this book is devoted to these two needs.

CHAPTER THREE
THE ATARI HARDWARE

We will now learn a little about the workhorse of our ATARI, the 6502 family of microprocessors. All real computing in our ATARI is done inside the chip known as the **CPU** (**C**entral **P**rocessing **U**nit), sometimes called the **MPU** (**M**icro **P**rocessing **U**nit). Different computers use different CPUs, the Z-80, the 8080, and the 6502 among them. The ATARI computers and many other popular machines use the 6502, or a modification such as the 6502A or 6502B, as their CPU. When we speak of programming in machine language or assembly language, we are really talking about programming the 6502 directly or indirectly.

In addition to the 6502 microprocessor, there are three additional specialized chips in our ATARI, and these are found in no other microcomputer. They are called ANTIC, POKEY, and GTIA (or in some of the first ATARIs, CTIA). They work in concert with the 6502 to produce the spectacular graphics and sounds that we have all come to associate with the ATARI computers. Their presence in only ATARI computers explains why frequently the same program run on an ATARI and some other microcomputer looks and sounds so much better on the ATARI. The uses for each of these chips and the ways to access them in our programs will be discussed later in this book.

Let's take a brief exploratory tour of the 6502, and learn a little about the way it works. The first thing that we may be surprised to

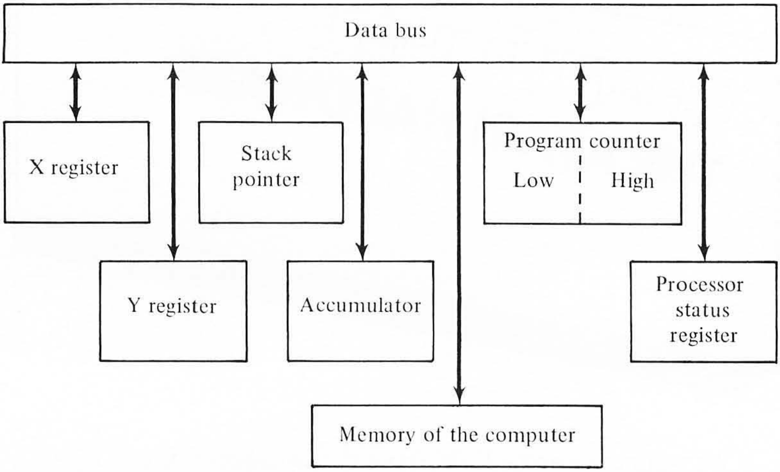


Fig. 3-1 Block diagram of a 6502 computer

learn is that this powerful computer of ours really only knows how to compare two numbers, or to add or subtract them! What happened to square roots? Division and multiplication? All the complex math we can do so easily in BASIC? Since these functions are really only combinations of comparisons, addition, and subtraction, we can easily teach our ATARI how to perform complex math, as we'll see later. Meanwhile, let's see how our computer works. There are six parts to each member of the 6502 family (Fig. 3-1), and we'll discuss each individually and then talk about how they work together.

THE ACCUMULATOR

The first part of this complex chip is the **accumulator**, usually called A in assembly language shorthand. This is the part that actually does the computing — the comparisons, the addition or subtraction. One way to think about the accumulator is to picture it as looking like the capital letter Y. You can stuff numbers into each of the top arms and operate on the numbers (add or subtract) to produce a result that can be pulled out of the bottom. The accumulator is unique in this respect, since it is the only place in the computer that can operate on two pieces of information at the same time. Let's think about a simple analogy in BASIC for a moment. If we

```
POKE 752,1
```

and then immediately follow that instruction with

```
POKE 752,0
```

we know that location 752 will now have the value of zero. That is, it cannot have both values simultaneously. We also know that we cannot add 12 to the value stored in memory location 752 directly. In order to do this in BASIC, the following program would be required:

```
10 I=PEEK(752)
20 I=I+12
30 POKE 752,I
```

What we did was pull out the value stored in memory location 752, increase it by 12, and put the new value back into memory location 752. How did we increase the value by 12?

We used the accumulator to increase the value of I by 12 in line 20. How we did this, and the exact instructions required for this manipulation, will be covered later. For now, it's enough to realize that the only part of our ATARI that can actually perform mathematical operations is the accumulator. Whenever we need to perform any math, we must follow the pattern shown above in the BASIC example; that is, we must load the value to be changed into the accumulator, change it, and store it back where we need it. This is an operation fundamental in assembly language programming, as we'll see shortly.

THE X AND Y REGISTERS

Our tour of the 6502 continues with the next two parts of the chip, the X and Y registers. These two storage locations are housed directly in the 6502 CPU, unlike the many other memory locations in our ATARI. They cannot be accessed directly from BASIC, but they are addressed frequently from assembly language. The registers can be used in either of two ways. The first is fairly simple and is exactly analogous to the BASIC POKE command. That is, we can use these two registers as simple storage locations to house information that we know we will need shortly. This is a fairly simple use of these two powerful registers. Their second use is as offset counters, or index registers. For instance, suppose we want first to access location 752, then 753, and then 754. In BASIC, we could do this in two ways:

```
10 A(1)=PEEK(752)
20 A(2)=PEEK(753)
30 A(3)=PEEK(754)
```

or

```
10 FOR X=0 TO 2
20 A(X)=PEEK(752+X)
30 NEXT X
```

The first way is usually referred to as the "brute-force" approach. It works all right as long as the number of items to be accessed is low, but if we just change the problem to require the accessing of 30 locations instead of 3, our program grows to be 30 lines long instead of 3. In the second example, the only thing that changes is the 2 in line 10. This is a more general and much more versatile solution to the problem posed. As you can see, we are using X as an offset from location 752. That is, the first time through the loop, we access location 752 + 0, or 752. The second time through the loop, we access location 752 + 1, or location 753. We are using the value of X as an offset from the base address of 752.

The X and Y registers of the 6502 can be used in exactly the same way, giving us a quick and easy way to access information stored in consecutive memory locations. Since so much information can be stored this way — arrays, strings, tables, screens, and the like — we have a very easy way of building information and of finding out what we've built.

THE PROGRAM COUNTER

Continuing our tour, we next encounter the program counter, or PC. The first thing that we notice about the program counter is that it's twice as big as the other registers in the 6502, 2 bytes wide instead of just 1; this is the only place in the whole computer which can act as a single 16-bit (2-byte) register. The program counter is responsible for remembering what comes next in your program. For instance, we have all learned that in a BASIC program, line 10 is executed before line 20, which in turn comes before line 30, and so on, and that each line of BASIC is converted to machine language before execution. How does the computer remember where it is and what machine language instruction comes next? The program counter tells it. The program counter is a 16-bit register because it must be able to point to every memory location in the computer. As we learned in Chapter 2, 16 bits are required in order to address 65,536 memory locations, so the program counter must be 16 bits wide. Remember, the program counter always points to the **next** instruction to be executed.

THE STACK POINTER

Before we continue our tour of the 6502, how about stopping for lunch? Here's a cafeteria — let's stop in here for a quick bite (byte?). First let's get a tray and silverware. Let's see . . . anything else we need before going through the line? Oh yes, a plate. We'll just grab one from this stack here. Notice how when we grab a plate from the top of the stack, the whole stack moves up one plate, so the next plate is now in the position ours was in before we grabbed it. It's a spring-loaded stack. We could keep taking plates off the top, and there would still be a plate in the same top position. The stack would be one plate shorter, and all of the plates in the stack would be one position higher, but the top plate would always be in the same position, until, of course, we ran out of plates.

Now that we've eaten, let's get back to our tour. The next register on the horizon is called the **stack pointer**. Sound familiar? Yes, it works just like the stack of plates we just saw in the cafeteria. Let's use a BASIC example again. Look at the following BASIC program:

```
10 GOSUB 40
20 PRINT "GOODBYE"
30 END
40 PRINT "HELLO"
50 RETURN
```

If we trace the flow of this program, we see that first we will print the word HELLO to the screen, and then we will print the word GOODBYE below it. The program will then end. How does this happen?

First, we go to the subroutine at line 40, where we print out the first word. Then we get to the RETURN in line 50, which causes line 20 to be executed. How does the computer know that line 20 should have been the next line executed after coming back from the subroutine? Aha! That's where the concept of a stack comes in. BASIC uses a run-time stack, just like the stacks we have been discussing. When the GOSUB statement in line 10 was executed, the first thing BASIC did was to push the line number and offset within that line onto its run-time stack. This stack is distinct from the 6502 stack, since in all 6502-based computers, page 1, memory locations 256 to 511 inclusive, is used as a stack. Both of these stacks work just like the cafeteria; if we then push additional addresses onto the stack, the first address will simply move down the stack as additional numbers are added (Fig. 3-2).

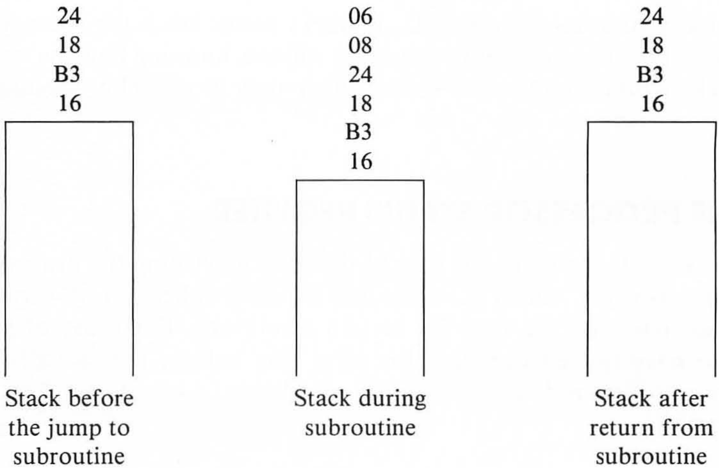


Fig. 3-2

The stack pointer, then, is the part of the 6502 which keeps track of what is currently on the bottom of the page 1 stack. We don't have to worry about how many values are on the stack, or how to add a value to the stack. The 6502 handles all of that overhead for us, just like BASIC takes care of its own stack without our worrying about it. The only thing we do have to worry about is that we don't try to stuff more than 256 numbers onto the stack. Since the maximum size of the stack is 256 numbers, if we put more numbers onto it, the first numbers that we pushed on will fall off the bottom and we'll lose them. Then, when we try to pull them back off to use them in some way, they won't be accessible, and we'll probably have a crash.

Now that we know how to get numbers onto the stack, how about getting them off again? In our BASIC example, we finished the subroutine with the RETURN statement in line 50. This statement tells BASIC to pull the top line number and offset off the stack, and RETURN to that location. That's how we get back to line 20, which is where we are supposed to be after the subroutine. Notice that we don't have to know anything about stacks in order to use a subroutine in BASIC. It works pretty much the same way in assembly language, although as we will see, knowing how the stack works is important to its several other uses in assembly language.

THE PROCESSOR STATUS REGISTER

We will complete our tour of the 6502 by visiting the **processor status register**, which is really just a 1-byte collection of various flags that the 6502 uses for certain conditions. For those of you who have not encountered the term **flag** before, it is a variable whose value indicates a certain condition. Let's use a BASIC example:

```
10 I=0
20 IF FILE=33 THEN I=1
30 . . .
```

In this example, we could check in line 30 to see if FILE=33 by checking the value of I. If I=0, then we know that FILE doesn't equal 33, but if I=1, then we know that FILE=33. In this example, I is a flag which gives us information about the value of FILE. In much the same way the seven flags in the processor status register of the 6502 give us considerable information about what's happening during our program. Each flag is a single bit in the single byte of this register (Fig. 3-3). Flags are usually known by their single-letter abbreviations, as follows:

Letter	Flag	Meaning
C	Carry	1 = true
Z	Zero	1 = result of zero
I	IRQ disable	1 = disable
D	Decimal mode	1 = true
B	Break command	
V	Overflow	1 = true
N	Negative	1 = negative

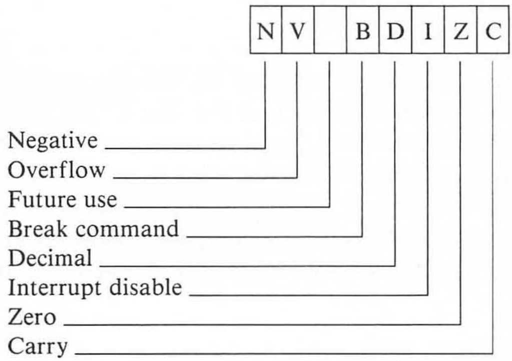


Fig. 3-3 Processor status register

THE CARRY FLAG

The **carry flag**, C, tells us whether or not the previous operation set the carry bit; that is, whether or not an addition summed to greater than 255. As a simple example, let's add 250 + 250. We can all easily calculate (perhaps with a little help from our ATARIs) that the answer is 500. However, this presents a bit of a problem in assembly language programming. Since we know that 255 is the largest 1-byte number we can have, how do we possibly represent the answer to this simple problem? Well, we can view the answer as 500 − 255 with a carry. That is, since the answer is larger than 255, we carry 1, and the answer is 245 with a carry of 1. But how can we tell the difference between an answer of 245 and an answer of 245 with a carry? Since we first set the carry bit in the processor status register to zero, and we add 250 plus 250 in the accumulator, we end up with 245 remaining in the accumulator. The carry bit is now 1, instead of zero, allowing us to calculate the true sum. If we add 240 + 5, we would again find 245 as the answer in the accumulator, but the carry bit remains zero, enabling us to distinguish between the two situations. Note that since each of the two numbers we add together must be less than 256, we will never run into the situation where the carry will have to be 2, so a 1-bit carry flag is sufficient for our needs. As we shall see, the carry bit is used in virtually all mathematical operations in assembly language.

THE ZERO FLAG

The **zero flag** tells us whether or not the previous operation yielded a result equal to zero. If it did, the Z flag equals 1. Therefore, if the Z flag starts out equal to zero, and we subtract 2 from 2, the accumulator will contain the value 0, and the Z flag will be 1. To determine whether or not something is equal to zero, we just have to operate on it in any of several possible ways and then look at the Z flag. We will see how useful this is in later chapters.

THE IRQ FLAG

IRQ stands for interrupt request. If you have read articles on any of the more advanced techniques possible on the ATARI computers, you are probably familiar with the term **interrupt** as in display list interrupt or vertical blank interrupt. Before you finish this book, these techniques will be easy for you to add to your own programs. The 6502 can be interrupted from its normal operations only if the I flag is equal to 0. If I is equal to 1, then normal interrupts are not possible. This fact will be important in later discussions of various interrupts used in the ATARI. For now, just remember that in order for interrupts to occur normally, the I flag in the processor status register must be equal to 0. If we set the I flag to 1, interrupts will not be allowed. We call this **masking** the interrupts.

THE DECIMAL FLAG

The 6502 has two modes in which it can operate, binary and decimal. The value of the **decimal flag**, D, in the processor status register determines which mode the processor is in. If this value is 1, all operations will be in the decimal mode, and if it is 0, they will be binary. In general, most operations in assembly language use binary math, but you have the ability to switch by toggling this flag.

THE BREAK FLAG

The **break flag**, or B flag, can be set and cleared only by the 6502 itself. The B flag cannot be altered by the programmer. It is used to determine whether an interrupt was caused by the 6502 instruction BRK, which stands for **BR**ea**K**. Since it cannot be set or reset by the programmer, the B flag has little function in a normal program, and in general is used only to determine program flow.

THE OVERFLOW FLAG

Although each byte consists of 8 bits, as we discussed in Chapter 2, in signed binary math the most significant bit is used to indicate the sign of the number; therefore, the largest signed number we can represent in 1 byte is 128. Here's a situation analogous to that requiring the carry flag discussed above. What happens if we try to add +120 and +120 together? The answer should be +240, but expressing this number requires the use of the most significant bit, which, in signed math, represents the sign, not part of the number. Therefore, if we are doing signed math, we somehow need a way of determining whether the math has overflowed into the sign bit. The **overflow flag**, V, is used to determine this. If the V flag is 1, overflow into the sign bit has occurred, and if it is 0, no overflow has occurred. We can therefore test this flag to be sure the number we have produced can safely be interpreted as a signed binary number. It is important to note this bit when doing signed math, and to allow a way for the program to deal correctly with such overflow, so it can still correctly interpret signed numbers, regardless of overflow.

THE NEGATIVE FLAG

The final flag in the processor status register is the **negative flag**, N. If this flag is 1, the previous operation yielded a negative result, and if the N flag is 0, the result was either positive or equal to zero. Note that we can then determine whether a number is zero or positive by testing the Z flag. Tests of the N, C, and Z flags represent the major methods for allowing for branching in an assembly language program, similar to IF...THEN logic in BASIC programs.

This concludes our brief tour of the 6502 chip, the heart of our ATARI computer. Now that we know the layout of the hardware, we can begin to learn the instructions necessary to program it.

MEMORY ALLOCATION SYSTEM

We have already discussed one aspect of memory allocation in computers using the 6502; that is, that the stack occupies a specific place in memory. Memory in a 6502 computer is divided into **pages**, each of which is 256 bytes long. You have probably already encountered the term page, especially in connection with the area of memory reserved for you, the programmer, by ATARI: page 6. Page 6 is the area of memory from $600 to $6FF, or in decimal nomenclature, from 1536 to 1791, and ATARI states that none of their software will ever require that space, so it is free for your use. Actually, this is not quite true, so be very careful when using this space. Page 6 is located, cleverly enough, directly above page 5; and the high half of page 5 is used by the ATARI computers for several purposes. There are certain conditions when you may overflow this area, and the overflow will be stored at the beginning of page 6, right on top of your carefully protected information. The moral: use page 6 with care, and be aware of the potential pitfalls.

Other pages of memory also have specific uses in 6502-based computers. The most important of these is page 0, the first 256 bytes of memory in the computer. Page 0 has particular significance to the assembly language programmer, since all access to this page is faster than access to anywhere else in the computer, and since certain operations can only be performed using page 0 locations. However, here we run into a major snag. Since this area is so important, you might expect to have it all for your use. Wrong! Since it's so important, ATARI used almost all of page 0 for their own use. In fact, if you have either the BASIC or the Assembler/ Editor cartridge in place, only 6 bytes of page 0 are available for your use! That's right, six. So we're going to learn a few tricks to make more of page 0 available, and we're going to make judicious use of the locations at our disposal.

We'll learn a lot about pages 2 to 5 ($200 to $5FF, or decimal 512 to 1535), which contain information needed by the operating system. The pages above page 6 are generally reserved for DOS. Memory that the assembly language programmer can safely use without running a risk of having programs overwritten by DUP.-SYS generally begins at $3200, or decimal 12800. Any cartridge which may be present generally starts at $A000 and continues up to $BFFF; after this, memory for the operating system goes all the way up to the top of memory, $FFFF. Many of these locations will also be discussed in detail in later chapters, but this outline serves as an introduction to the memory allocation system in your ATARI, and, in broad strokes, paints a picture of what goes where. The details will be filled in as we proceed.

PART TWO
LEARNING ASSEMBLY LANGUAGE

CHAPTER FOUR
NOMENCLATURE AND THE INSTRUCTION SET

A WORD ABOUT NUMBERS

Before we can discuss the 6502 instruction set itself, we need to briefly discuss some shorthand used in all 6502 assemblers. This will allow us to write numbers and abbreviations properly, and let us understand one another.

Whenever a number is used in an assembly language instruction, it must be preceded by a number sign, #. For example, if we refer to the number 2, we need to write #2. Then the assembler can distinguish between a number and a specific address inside the computer. When the number is preceded by the # sign, the assembler knows that you mean a number, and when a number appears alone, the understanding is that you mean an address. Take the following examples, written in English, for instance:

```
add 2 to SUM                      SUM + #2
add the contents
of memory location 2
to SUM                            SUM + 2
```

The single biggest mistake that beginning assembly language programmers make is to use numbers for addresses and addresses for numbers. This will completly destroy any program, and if you're not familiar with assembly language programming, you can look at a printout of the program for days without spotting the error.

The second convention used in assembly language programming involves number base. Whenever a number appears either alone or preceded only by the # sign, the assembler knows that you mean base 10, the decimal system; for example,

```
SUM+#11
```

The assembler interprets this to mean that the decimal number 11 should be added to the value of SUM. Similarly, we could write this:

```
SUM+11
```

The assembler interprets this to mean that the contents of memory location 11 (in the decimal numbering system) should be added to the value of SUM.

When we want to use the hexadecimal numbering system, we precede the number with a dollar sign; for example:

```
SUM+$11
```

This instruction means to add the value of SUM to the contents of memory location $11 (which is location 17 in the decimal system). Things get somewhat more complicated when we refer to a hexadecimal number. We must first tell the assembler that a number is coming, and then tell it that this number is in the hexadecimal system. Our example now looks like this:

```
SUM+#$11
```

This instruction means to add the hexadecimal number $11 (decimal 17) to the value of SUM. It cannot be misinterpreted by the assembler. Unfortunately, it certainly can be mistakenly written in a wide variety of forms by the novice to assembly language programming. So, to be forewarned is to be forearmed. These types of mistakes in writing assembly language programs are very common, and, if your first programs do not work, you should check for these types of mistakes first, before looking for complicated errors of logic.

A third type of number recognized by most assemblers is rarely used, but when it's needed, you'll be glad it's available. This is the binary system, which is usually prefaced by a percent sign, %; for instance,

```
#%11010110
```

There can be no confusion about the interpretation of this number, since the % sign clearly labels it as a binary number. Furthermore, the decimal number 11,010,110 is much too large to be directly addressed by the 6502-based computers.

To review, the # sign identifies the term following it as a number, to distinguish it from an address. The $ sign identifies the term following it as a hexadecimal term, and it follows the # sign, where a hexadecimal number is meant. The % sign identifies the next term as a binary term, and also follows the # sign in the case of a binary number. When neither the $ sign nor the % sign precedes the term, the decimal system is understood.

THE 6502 INSTRUCTION SET

Each instruction in the 6502 instruction set is described in detail in Appendix 1. We will briefly discuss the instructions here to familiarize you with the nomenclature and the use of the instructions. Each instruction is a three-letter abbreviation of the full name of the instruction. This abbreviation is called a **mnemonic**, and once learned, is fairly easy to remember. We'll cover the way these instructions address memory in Chapter 5.

In this section, we will discuss the instructions in groups, concentrating on how the instructions can be used in programming.

THE LOAD INSTRUCTIONS

There are three instructions in this group:

LDA **LoaD** the **A**ccumulator
LDX **LoaD** the **X** register
LDY **LoaD** the **Y** register

These instructions are in some respects similar to the PEEK instruction in BASIC. The PEEK instruction retrieves the value stored in a specific memory location. Any of the LOAD instructions can also be used to retrieve a value from memory, as in the following example:

```
LDA $0243
```

This command takes the value previously stored in the memory location with the address $0243, and places a copy of that value into the accumulator for further manipulation. Note particularly the use of the word **copy** in this statement. Like the PEEK command in BASIC, the LOAD instructions in assembly language programming do not change the value stored in the location from which the load takes place. Location $0243 contains the same value before and after the LDA instruction is executed; however, the value contained in the accumulator changes as a result of this instruction. We could have chosen to transfer this value from location $0243 to either the X or the Y register; in this case, the above line would have read either LDX $0243 or LDY $0243 respectively.

Since we already know that all calculations such as addition and subtraction are done in the accumulator, one use of the LDA instruction becomes obvious. There are, of course, many other uses for this instruction. The other two LOAD instructions, LDX and LDY, are used to load either of the registers with a specific value, usually prior to using the register in some other operation, such as counting. Many examples of the LOAD instructions will be discussed throughout the book.

THE STORE INSTRUCTIONS

As we discussed, the BASIC PEEK command and the assembly language LOAD instructions are somewhat similar. In assembly language, we also have commands analogous to the BASIC POKE command, the STORE commands. Since there are three LOAD commands, it is not surprising to find that there are also three STORE commands:

STA **ST**ore the **A**ccumulator
STX **ST**ore the **X** Register
STY **ST**ore the **Y** Register

A typical line of assembly language code using these instructions might appear as follows:

```
STX $0243
```

This instruction copies whatever value was previously stored in the X register into memory location $0243. The analogy with the BASIC POKE command is obvious. As with the LOAD instructions, the value stored in either the accumulator or the registers, depending on which instruction is used, is not changed by the execution of the instruction. Therefore, if you wanted to store the number 8 into four different memory locations, the following code could be used:

```
LDX #8   ;first, load it in
STX $CC  ;the first location
STX $CD  ;the 2nd location
STX $12  ;the 3rd location
STX $D5  ;and we're done
```

Note especially that we don't have to reload the X register with 8 before each store command. The value remains there until we change it. Of course, we could just as easily have used either the accumulator or the Y register to accomplish the above goal in the same fashion.

One very common use of the LOAD and STORE instructions is to transfer the values stored in one or more memory locations into different locations. For example,

```
LDA $5982  ;get the 1st value
STA $0243  ;transfer it
LDA $4903  ;get the 2nd
STA $82    ;and so on...
```

In Chapter 7, we'll see how to use this type of routine to write subroutines which can speed up your BASIC programs amazingly.

TRANSFER OF CONTROL INSTRUCTIONS

Two types of instructions cause program control to shift from one place in the program to another. These are the JUMP instructions and the BRANCH instructions.

THE JUMP INSTRUCTIONS

For the purposes of this discussion, we have grouped two instructions into this category:

JMP **JuMP** to a specific address
JSR **J**ump to a **S**ub**R**outine

These two instructions are analogous to the BASIC commands GOTO and GOSUB, respectively. Both instructions result in unconditional transfer of program flow. Here's an example of the JMP instruction:

```
       JMP SUB1   ;GOTO SUB1
SUBO   LDA #1     ;to inhibit cursor
       STA 752    ;store a 1 here
SUB1   LDA #0     ;to reset cursor
       STA 752    ;store a 0 here
```

In this example, the cursor will never be inhibited, since whenever the program gets to the JMP instruction, the line labeled SUB1 is executed next. This transfer of control is **unconditional**; that is, it will happen every time. The 2-line routine labeled SUB0 will never get executed.

In contrast, let's look at an example of the JSR instruction:

```
        JSR SUB1   ;GOSUB SUB1
SUB0    LDA #1     ;to inhibit cursor
        STA 752    ;store a 1 here
        JMP SUB2   ;to avoid SUB1
SUB1    LDA #0     ;to reset cursor
        STA 752    ;store a 0 here
        RTS        ;like BASIC's RETURN
SUB2    ...        ;more code ...
```

In this routine, we JSR to the subroutine labeled SUB1. The program then executes the lines in order, until an RTS (**R**e**T**urn from **S**ubroutine) instruction is encountered. Program control then reverts to the line following the JSR that sent control to the subroutine in the first place. The RTS instruction is the assembly language counterpart to the BASIC RETURN command, which also marks the end of a subroutine. In the example, first SUB1 will execute, and then SUB0 will execute. The JMP instruction following SUB0 simply prevents SUB1 from executing a second time. There is another instruction in assembly language which is similar to the RTS instruction, the RTI (**R**e**T**urn from **I**nterrupt). This instruction is used at the end of an interrupt routine to return control to the main program, like the RTS instruction. We'll discuss interrupts at great length in later chapters.

THE BRANCH INSTRUCTIONS

In contrast to the two unconditional transfer of control instructions just discussed, the 6502 has an extensive set of **conditional** transfer of control instructions. These can be compared to the IF...THEN construction of BASIC:

```
IF X=5 THEN GOTO 710
```

This statement will transfer control to line 710 only if X is equal to 5. If it equals any other value, program control will shift to the next line of code following the IF statement. In a sense, by coding this line we have allowed the computer to decide what to do, depending on conditions we have established; and we've set up a conditional transfer of control. These are the branch instructions of the 6502 instruction set:

BCC **B**ranch on **C**arry **C**lear
BCS **B**ranch on **C**arry **S**et
BEQ **B**ranch on result **EQ**ual to zero
BMI **B**ranch on result **MI**nus
BNE **B**ranch on result **N**ot **E**qual to zero
BPL **B**ranch on result **PL**us
BVC **B**ranch on o**V**erflow **C**lear
BVS **B**ranch on o**V**erflow **S**et

Each of these instructions depends on the value of one of the flags in the processor status register. Whether or not the branch is taken depends on the value of that flag at that time, so these are clearly conditional transfer of control instructions. Let's look at a simple example to see how these instructions work:

```
        LDA #0    ;initialize
        BCC SUB4  ;branch if carry clear
        LDA #1    ;if not
SUB4    STA $0243 ;store the value here
```

In this routine, the value stored into memory location $0243 depends on the condition of the carry flag in the processor status register at the time the branch instruction is executed. If the carry flag is set (equal to 1), then the branch is not taken, and the accumulator is loaded with the value 1 before the STA command. If the carry flag is clear (equal to zero), the branch is taken, the accumulator is not changed, and the value 0 is stored into memory location $0243. The BCS instruction is the opposite of the BCC instruction: the branch is taken if the carry bit is set and is not taken if the carry bit is clear.

The BEQ and BNE instructions depend on the value of the zero flag in the processor status register, rather than on the value of the carry flag. If the zero flag is clear, the BEQ branch is not taken, but the BNE branch is taken. If the zero flag is set, the BEQ branch is taken, and the BNE is not. For instance, we do not take the branch here:

```
LDA #0    ;sets the zero flag
BNE SUB4  ;branch is not taken
```

but we would have branched to SUB4 if we had written this:

```
LDA #1    ;clears the zero flag
BNE SUB4  ;branch is taken
```

The overflow flag is used to determine the outcome of the BVC and BVS instructions in an analogous fashion. Similarly, the negative flag determines the outcome of the BMI and BPL instructions. If previous instructions produce a negative answer, then a branch based on the BMI instruction is taken. If this answer is either positive or equal to zero, the BPL instruction is taken. Used appropriately, these eight instructions, which depend on the values of four of the flags in the processor status register, can give extremely fine control over program flow in assembly language programs, as we shall see in subsequent chapters.

PROCESSOR STATUS REGISTER INSTRUCTIONS

These instructions directly manipulate the flags in the processor status register:

CLC **CL**ear the **C**arry flag
CLD **CL**ear the **D**ecimal flag
CLI **CL**ear the **I**nterrupt flag
CLV **CL**ear the o**V**erflow flag
SEC **SE**t the **C**arry flag
SED **SE**t the **D**ecimal flag
SEI **SE**t the **I**nterrupt flag

These instructions perform the indicated operations directly on the flags of the processor status register, and their operation, which is self-explanatory, is further described in Appendix 1.

ARITHMETIC AND LOGICAL INSTRUCTIONS

We will place all of the calculating instructions of the 6502 into this group of instructions.

ADC	**AD**d with **C**arry
AND	the logical **AND** instruction
ASL	**A**rithmetic **S**hift **L**eft
BIT	test **BIT**s of memory with the accumulator
EOR	**E**xclusive **OR**
LSR	**L**ogical **S**hift **R**ight
ORA	logically **OR** memory with the **A**ccumulator
ROL	**RO**tate **L**eft
ROR	**RO**tate **R**ight
SBC	**S**u**B**tract with **C**arry

These are all complex instructions; for a detailed explanation of them, please see Appendix 1; for a brief discussion, read on.

The ADC instruction is the fundamental addition instruction of the 6502. It takes the sum of the value stored in the accumulator, plus the carry bit in the processor status register, plus the number addressed by the ADC instruction itself. For instance, let's add the contents of memory location $0434 to the contents of memory location $0435, and store the result in memory location $0436:

```
CLC       ;clear carry bit first
LDA $0434 ;get 1st number
ADC $0435 ;add 2nd number
STA $0436 ;store the result
```

We'll be using ADC frequently throughout the remainder of this book. Its counterpart is the subtraction instruction, SBC. SBC subtracts the value addressed from the value stored in the accumulator, using the carry bit of the processor status register if a borrow is needed to perform the subtraction. To subtract the same values we added above, we would write this:

```
SEC       ;in case we need to borrow
LDA $0434 ;get 1st number
SBC $0435 ;subtract 2nd one
STA $0436 ;store the result
```

There are four SHIFT instructions in this group, ASL, LSR, ROL, and ROR. These instructions all shift the bits of a number, but in different ways. The two ROTATE instructions use the carry bit of the processor status register, and literally rotate the 8 bits of the number addressed through the 9 positions (8 in the number itself, and 1 from the carry bit). Pictorially, this looks like the following example:

```
ROR $0434 ;rotate right contents of $0434
```

START

in $0434	**in C**
10110100	1

END

in $0434	**in C**
11011010	0

As you can see, each bit rotated one position to the right, with bit 0 ending up in the carry bit and the former carry bit ending up in bit 7 of location $0434.

The ROL instruction simply reverses the rotation, to the left instead of the right. The two SHIFT instructions ASL and LSR work in almost the same way, except that although the end bit winds up in the carry bit as above, zero, instead of whatever was in the carry bit, is always rotated into the number.

These four SHIFT instructions are used to multiply or divide by powers of 2, since by rotating bits to the left, we double a number, and by rotating bits to the right, we effectively divide a number by 2. There are cautions to observe when using these instructions for this purpose, however, as will be described in Appendix 1.

The three logical instructions, AND, EOR, and ORA, are simply three ways of comparing two numbers bit by bit. They take the binary forms of the two numbers being compared, and, depending on whether both numbers contain a one or a zero in each bit, produce different results. The AND instruction says "If both bits are 1, the result will also have a 1 in that position. If not, the result will have a zero in that position." The EOR instruction says "If one, and only one, of the numbers has a 1 in that position, the result will also have a 1 in that position. If both numbers have a 1 or both contain 0, the result will have a zero in that position." Finally, the ORA instruction says "If either or both numbers have a 1 in this position, the result will also have a 1 in this position." These three logical instructions are used in a wide variety of ways. ORA is most commonly used to set a specific bit in a number, EOR to complement a number, and AND to clear a specific bit of a number. If you are unfamiliar with these three logical operations, see Appendix 1 for further details.

The final instruction of this group is BIT, which is a testing instruction. BIT sets the negative flag of the processor status register equal to bit 7 of the number being tested, the overflow flag equal to bit 6 of the number being tested. BIT also sets the zero flag, depending on the result of ANDing the number being tested with that stored in the accumulator. This instruction tests several aspects of a number all at once. Note that the number in the accumulator is unchanged by the BIT instruction. By following this instruction with one of the BRANCH instructions we have already discussed, we can cause an appropriate branch in the execution of the program.

6502 MANIPULATION INSTRUCTIONS

Like the LOAD and STORE instructions we discussed earlier, the following instructions involve interchanging information from one part of the computer to another:

PHA **P**us**H** the **A**ccumulator onto the stack
PHP **P**us**H** the **P**rocessor status register onto the stack

PLA **P**u**L**l from the stack into the **A**ccumulator
PLP **P**u**L**l from the stack into the **P**rocessor status register
TAX **T**ransfer the **A**ccumulator to the **X** register
TAY **T**ransfer the **A**ccumulator to the **Y** register
TSX **T**ransfer the **S**tack pointer to the **X** register
TXA **T**ransfer the **X** register to the **A**ccumulator
TXS **T**ransfer the **X** register to the **S**tack pointer
TYA **T**ransfer the **Y** register to the **A**ccumulator

The functions of these instructions, too, are self-explanatory. They are used to interchange information between the various registers of the 6502, or to store information on the stack for later retrieval. The PHA and PLA instructions are used frequently to pass information between a BASIC program and a machine language subroutine, as we shall see.

INCREMENTING AND DECREMENTING INSTRUCTIONS

Instructions in this group can increase or decrease by 1 the value contained either in a specific memory location, or in one of the 6502 registers:

DEC **DEC**rement a memory location by one
DEX **DE**crement the **X** register by one
DEY **DE**crement the **Y** register by one
INC **INC**rement a memory location by one
INX **IN**crement the **X** register by one
INY **IN**crement the **Y** register by one

These instructions are straightforward. Here is an example of their use:

```
LDA #3    ;start with 3
STA $0243 ;stored here
INC $0243 ;now it's a 4
INC $0243 ;now it's a 5
DEC $0243 ;now it's a 4 again
```

Note that there is no incrementing or decremeting instruction which operates on the accumulator. To increase or decrease a number in the accumulator, we must use the ADC or SBC instruction. Therefore, if a simple increment or decrement is required, it is easier to load a number into either the X or Y register, rather than into the accumulator, and then simply use the appropriate increment or decrement instruction.

THE COMPARE INSTRUCTIONS

Three instructions allow comparisons to be made between two values. These instructions condition various flags in the processor status register, depending upon the outcome of the comparison:

CMP **C**o**MP**are the accumulator with memory
CPX **C**om**P**are the **X** register with memory
CPY **C**om**P**are the **Y** register with memory

The way each of these affects the processor status register is described fully in Appendix 1, but a simple example is given here to demonstrate the use of the COMPARE instructions:

```
LDA $0243    ;get the 1st number
CMP $0244    ;compare it to the 2nd
BNE SUB6     ;go to SUB6 if $244 >$243
LDA #1       ;else, do this
```

THE REMAINING INSTRUCTIONS

Two final instructions do not easily fall into any grouping. These are the BRK (**BR**ea**K**) and the NOP (**N**o **OP**eration) instructions. The BRK instruction is used primarily in debugging your program once you've written it. It causes the program being executed to stop, and is somewhat similiar in this regard to the BASIC STOP instruction. The NOP instruction does nothing; its primary function is to reserve space in a program for future changes which may need to be made. It may be necessary to reserve this space, since frequently in an assembly language program, the exact memory location occupied by an instruction may be critical, and NOP instructions in the code can be replaced by functional commands without changing the location in memory of the instructions that follow it.

This completes our short introduction to the instruction set of the 6502. As we have already stated, details on any of these instructions can be found in Appendix 1. It is strongly advised that beginners to assembly language read Appendix 1 thoroughly. If you are already familiar with the instruction set, this short discussion should have refreshed the instructions in your mind, and you are ready to proceed.

CHAPTER FIVE
ADDRESSING TECHNIQUES

INTRODUCTION TO ADDRESSING TECHNIQUES

To begin our discussion of addressing techniques, let's first define the term. As used throughout this book, the term **addressing** refers to the way we tell the 6502 what memory location we wish to operate on. For example, BASIC has two addressing modes. The first is a direct mode in which the memory location (address) in question is specified directly, for example:

```
20 POKE 752,1
25 V=PEEK(764)
```

Line 20 tells the computer to put the value 1 into memory location 752. Line 25 tells the computer to look directly into memory location 764, get the value stored there, and then store this value into a variable called V. This direct accessing of one particular memory location is called **direct addressing**.

The second system used in BASIC is more subtle, and is also implied by line 25 above. When we tell the computer to take the value from memory location 764 and store it into a place called V, we, as programmers, don't care where V actually is. It's enough that the computer knows where V is stored, and that it knows how to retrieve the correct value when we refer to V from this point on. We'll call this form **indirect addressing**.

Both of these BASIC modes have counterparts in assembly language, and we'll discuss them later in this chapter. Many other addressing modes are also available from assembly language: let's first use a nonprogramming example to see why it is advantageous to be able to use more than one or two addressing modes.

Imagine a very large apartment building with thousands of apartments, so large that it dwarfs anything else in the world. It has 256 floors, making it the world's tallest building by far, and each floor has 256 apartments. In fact, this building is so large that it has its own postal system. Now let's think for a moment about how we tell the poor letter carrier how and where to deliver internal mail to the residents of this huge building. By the way, this building is named the 6502 Building, since it reminded the architects of a computer based on the 6502 chip, with 256 pages of memory each containing 256 memory locations.

We can, of course, give the letter carrier a specific letter with instructions to deliver it to apartment 5-004. The carrier would then take the elevator to the fifth floor and slide the letter into the slot in the door marked 004. Since we gave an absolute address, which didn't vary, or depend on any other information, we could refer to this as **absolute addressing**.

The ground floor of our building is occupied by many of the offices which are required to keep a building of this size running, and of course they'll need to receive mail also. If the address we specify on the letter is 0-032, we have a special case of the absolute address. The letter carrier doesn't need to use the elevator to reach a ground-floor address, so this letter can be delivered much more quickly. In fact, it has even become standard for residents to omit the first zero if the mail is destined for the ground floor, floor 0, and simply put 032 on the envelope. Our letter carrier knows this is a quick delivery and runs it right over. Letters to these offices are very important and must be delivered immediately. Since this is a special case of the absolute address, in which we don't even specify the floor, we'll call it **zero floor addressing**.

These two addressing systems both specify the exact absolute address on the envelope. Now suppose that we want to send a bulk mailing to everyone on floor 123. We could, of course, address each of the 256 letters individually, but this would take a long time. Instead, we might simply hand the letter carrier 256 letters and a note with instructions to deliver one of the enclosed letters to each apartment on floor 123. The carrier would then take the elevator to the 123d floor, and then walk along the halls, dropping letters in the slots on the doors and counting 1, 2, 3.... With any luck at all, the carrier would get to apartment 123-256 with one letter left, and deliver it. This type of address requires a count offset from the floor address, in order to get the apartment number. For instance, the 12th letter must go to apartment 123 plus 12, or number 123-012. Our count is an index that tells us which apartment on this floor we have reached; we'll refer to this type of addressing as **indexed addressing**. We shall see that several different indexed addressing modes are used in the 6502 building.

If we lived in apartment 230-042, we could stop the letter carrier who delivered our mail, and request that a letter be delivered to an apartment five doors down from us. In this case, we don't have to mention the apartment number; the letter carrier can figure it out. The apartment to which this letter is delivered depends on the apartment from which it was sent. In our example, the letter ends up in apartment 230-047, but if the same message were given to the letter carrier by the owner of apartment 024-128, then the letter would be delivered to apartment 024-133. Therefore, this addressing system is relative to the address of the originating apartment; we'll call this mode **relative addressing**.

In addition to the addressing modes we have already discussed, our letter carrier must understand a whole set of instructions. For instance, the carrier must check to see if the postage has been put on each envelope. We don't have to specify that the letter carrier does this: this function is implied by the instruction. There are several **implied addressing** modes in the 6502 Building, as we shall see.

This example has shown what addressing modes are, and why there are a number of different modes available to make the task of addressing easier. Let's now leave our building, get back to our ATARIs, and discuss the various addressing modes available in the 6502.

MEMORY ADDRESSING MODES OF THE 6502

The 6502 microprocessor can address memory in 13 different ways, and we'll examine each of these now. For the first eight of these addressing modes, we'll use the LDA instruction discussed in Chapter 4, although many of the other instructions can also utilize these addressing modes. See Appendix 1 for details.

IMMEDIATE MODE

The first addressing mode is called the **immediate mode**, and specifies that we want to load the accumulator with the number which follows. For example,

```
LDA #$4F
```

tells the computer to load the accumulator with the number $4F, or decimal 79. The same instruction could be written like this:

```
LDA #79
```

Having multiple numbering systems available doesn't make things harder, just more versatile.

Note that any instruction written in the immediate addressing mode will result in 2 bytes of machine language code, 1 for the instruction and 1 for the number. Furthermore, we cannot specify a number larger than 255, since this is the largest number that can be coded in 1 byte. The 6502 is an 8-bit microcomputer, and now you know what that means — only 1 byte (8 bits) can be operated on at one time. A 16-bit microcomputer can load and operate on numbers up to 65,536, since this is the largest number which can be coded in 2 bytes (16 bits).

One further concept which can be covered here is the length of time it takes the computer to complete each instruction. We refer to this in terms of the number of cycles the instruction takes to execute. One **cycle** is the shortest time period the computer deals with, and the 6502A in your ATARI has a cycle time of 560 nanoseconds. That's 0.00000056 seconds! The immediate mode LDA instruction we just discussed requires 2 cycles to execute, or 0.00000112 seconds. That's about one microsecond! Now you can begin to see how fast your ATARI can really be.

ABSOLUTE MODE

The second addressing mode we'll discuss is the **absolute mode**, used when you want to load the accumulator from a specific, known memory address. The form of the command is:

```
LDA 315
```

which tells the computer to load the accumulator from memory location decimal 315. As for the immediate mode discussed above, the same instruction could have been written like this:

```
LDA $13B
```

If memory location 315 contains the value 243, for example, then the accumulator will also contain the value 243 following the execution of this statement. Note that LDA does not change the value stored in memory location 315; it just copies what was there into the accumulator. It's just like this BASIC statement:

```
10 A=PEEK(315)
```

Here we copy the contents of memory location 315 into the variable called A. Location 315 is unchanged following either the assembly language or BASIC instruction.

The absolute addressing mode produces 3 bytes of machine language code, since this instruction must be able to load the accumulator from anywhere in memory. That is, to code for any address above 255 requires 2 bytes, and we also need 1 byte for the instruction. By the way, the 6502 family of microprocessors addresses memory in low byte-high byte order, the reverse of some other popular microprocessors. For example, take the following instruction:

```
LDA $2F3C
```

When this assembly language instruction is assembled — translated into machine language by an assembler — the code for LDA in the absolute addressing mode, $AD, comes first, and the address comes next, in low-high order:

```
LDA $2F3C   becomes  AD3C2F
```

Reading printouts of assembler output can sometimes be confusing, but after a little practice, it will seem natural to you.

ZERO PAGE MODE

The next addressing mode that we'll discuss is called the **zero page mode**. This mode is used to load the accumulator from an address in the first 256 bytes of memory, page zero. Since no address on this page can be more than 1 byte long, the zero page absolute addressing system requires only 2 bytes, 1 for the instruction, and 1 for the address. For example,

```
LDA $2D
```

which could be written

```
LDA 45
```

tells the computer that the programmer wants to load the accumulator from an address which is on page zero. This instruction will be coded appropriately by any assembler. It is of interest to learn that the zero page addressing mode requires only 3 cycles to execute, in contrast to 4 for the absolute addressing mode. Therefore, in a program which requires maximum speed, assembly language programmers use page zero whenever possible. But remember that very few such locations are available under most conditions in your ATARI. If either the BASIC or Assembler/Editor cartridges is in place in your ATARI, only six page zero locations are available for use by an assembly language program. Having only six page zero locations sounds very hard to deal with, but some other popular microcomputers leave only two available, so we have a virtual embarassment of riches in the ATARI!

ZERO PAGE INDEXED MODE

The next addressing mode we'll discuss is called the **zero page indexed**, or the **zero page, X** mode, and is the first addressing mode to be discussed that uses the X register as an offset register for addressing. In this addressing mode, the value the X register has at the moment the instruction is encountered is added to the value of the specified address in order to arrive at the final address to be used. As an example, suppose the X register has the value 5 stored in it, and we encounter this instruction:

```
LDA $43,X
```

We already know that the first part of this instruction, if seen by itself, would mean to load the accumulator from the hexadecimal address $43, on page zero. To arrive at the correct loading address in this case, we simply add 5, the value contained in the X register, to the base address $43 specified in the instruction, and arrive at the hexadecimal address $48. Therefore, this instruction currently means to load the accumulator from the hexadecimal address $43 + 5, or $48.

Note the use of the word currently. This addressing mode is the first we have encountered in which an instruction does not always mean the same thing each time we see it. For instance, the X register might contain the value 2 when we encounter the same instruction:

```
LDA $43,X
```

Now we would not load from hexadecimal address $48, but rather from $45 ($43 + 2 = $45).

It should be apparent that if we first retrieve a value from one zero page address and then want to retrieve a value from the next-higher zero page address, by using this zero page, X addressing mode we could increase either the base address or the X register. That is, we could keep the value stored in the X register at 2 by writing this:

```
LDA $44,X
```

Or we could increase the value stored in the X register to 3, by writing this:

```
LDA $43,X
```

These two examples seem trivial, and you may well ask what difference it could possibly make to prefer one mode over another; but we shall see later that the second method, increasing the X register and keeping the instruction constant, is vastly preferable for several reasons. For now, it is enough to realize that there are several ways to accomplish the same end, and that one may be the best way, even if we are not yet sure why.

The zero page, X addressing mode requires only 2 bytes when converted to machine language and requires 4 machine cycles to execute. It is therefore slightly slower than the direct zero page addressing mode, which requires only 3 machine cycles. This sacrifice in speed is the price paid for the versatility and power gained. Of course, in applications which require pure speed, it may be necessary to take this slightly slower execution time into account, or to sacrifice the power of this instruction and use only zero page addressing.

THE ABSOLUTE INDEXED MODES

The next two addressing modes are so similar that they will be discussed together. They are relatives of the mode just discussed, but they are applicable to any address in the computer, not just zero page addresses. These are the **absolute, X** and **absolute, Y** addressing modes, often referred to as **absolute indexed** addressing modes. They work by adding the contents of the X or the Y register, respectively, to the base address referred to in the instruction. For instance, if the X register contains the value 3, then the instruction

```
LDA $0342,X
```

loads the accumulator from memory location $0345, since $0342 + 3 = $0345. In an analogous fashion, if the Y register contains the value 4, then the instruction

```
LDA $0347,Y
```

loads the accumulator from memory location $034B, since $0347 + 4 = $034B.

Since we need to address the entire memory space of the computer with these two instructions, they are both 3-byte instructions. They each require 4 machine cycles to execute, which is very interesting to us, since an absolute LDA instruction also requires 4 machine cycles to execute. Here's a case in which we're getting something for nothing — increased power and versatility at no increase in execution speed! Well, **almost** no increase. You knew there had to be a catch somewhere. Here's the problem. In the case where the base address plus the offset add together to produce an address on a page higher than that referred to by the base address, the 6502 requires 1 more machine cycle to correct for this. For example, suppose the value in the X register is 4, and we encounter this instruction:

```
LDA $05FF,X
```

The base address in this example is located on page 5; in fact, it is the last address on page 5. The offset, 4, if added to this address, results in the value $0603, so this instruction means to load the accumulator from memory location $0603. However, we can see that this location is on page 6, and the base address is on page 5. Although the 6502 can handle this problem, it takes somewhat longer — in fact, 1 cycle longer — to correct for this crossing of a page boundary. Any addressing modes in the 6502 which involve crossing such a page boundary require 1 extra cycle to execute. In general, this should not be a problem, and the computer will take care of it for us; but in cases where precise timing is critical, the programmer should be aware of such situations and make provisions for the extra time these instructions will require.

TWO INDIRECT MODES

The last two methods of addressing which will be illustrated using the LDA instruction are both indirect addressing systems. These two systems are frequently confused because their names are so similar, but their uses are quite distinct. We will find ourselves using one quite frequently and the other hardly at all. The first is called **indirect indexed addressing**, and here is the form of its operation:

```
LDA ($43),Y
```

The parentheses in this instruction indicate that this is an indirect addressing mode. The instruction can be interpreted as follows:

1. On page zero, in locations $43 and $44, find a 2-byte value stored.
2. Interpret this 2-byte value as an address in memory.
3. To this address, add the offset value contained in the Y register. This sum is the address to access for this operation.
4. Load the accumulator from this calculated address.

Whew! Seems pretty complicated, doesn't it? Let's take a simple example and work our way through it slowly.

First, let's assume that we have stored the value #$53 (this is the hexadecimal number 53, remember?) in memory location $43 and the value #$E4 in memory location $44. Furthermore, let's assume that the Y register contains the value 6. Pictorially, this is the situation:

Location	Contents
$43	#$53
$44	#$E4
Y register	#6

Now we encounter the instruction

```
LDA ($43),Y
```

The 6502 first looks at memory locations $43 and $44 and takes the values stored there as an address. In this example, it finds the values #$53 and #$E4, and, since it knows the first byte is the low value of the address, and the second is the high value, it realizes that the address referred to is $E453. The 6502 then adds the offset value, 6, obtained from the Y register, to this address, and calculates the address to be accessed to be $E453 + 6 = $E459. Finally, it executes the LDA instruction, and loads the accumulator from memory location $E459.

Although this seems like an extremely cumbersome and complicated way of calculating an address, we shall see how important and versatile this instruction really is. In fact, many applications we will use would not be easy without this addressing mode.

Since the indirect indexed addressing mode requires page zero, it utilizes only 2 bytes for the instruction. However, the calculations involved are fairly complicated, as we have seen, so this addressing mode requires 5 machine cycles to execute.

The last addressing mode to be discussed here is called the **indexed indirect mode**, and we can immediately see why it is often confused with the mode just discussed, the indirect indexed mode. However, its use is completely different. A typical instruction written in this mode follows:

```
LDA ($43,X)
```

We can see that this mode uses the X, rather than the Y register, and that the entire operand is enclosed within parentheses. This instruction would be interpreted by the 6502 as follows:

1. Add the value stored in the X register to the base address, $43 (e.g., if X = 4, then this sum equals $47).
2. This sum is then interpreted as another zero page address (in this example, the second zero page address is $47).

3. Find the 2-byte value stored at this calculated address ($47,$48) and interpret it as a new address (see below for a discussion of an example).
4. Load the accumulator from this new address.

Again, let's take an example. We'll assume that we have previously stored the value #$E4 in memory location $48, the value #$53 in memory location $47, and the value 4 in the X register. Pictorially, we have this:

Location	Contents
$47	#$53
$48	#$E4
X register	#4

Then we encounter this instruction:

```
LDA ($43,X)
```

We add the contents of the X register to the base address specified and obtain $43 + 4 = $47. We then look in memory locations $47 and $48, and interpret the 2 bytes there as a new address, $E453 (remember, low byte first and then high byte). Finally, we execute the instruction, loading the accumulator from memory location $E453. This operation requires 6 machine cycles and is therefore the slowest instruction we have yet encountered. Since it requires zero page addressing, it needs only 2 bytes per instruction. As was mentioned above, this instruction is seldom used. Its primary use is for establishing a table on page zero and then accessing this table to provide addresses elsewhere in memory. However, your ATARI has limited zero page space available for your use, especially when either the BASIC or Assembler/Editor cartridge is in place; we generally don't have room to construct such a table on page zero, and we use other addressing modes to construct such tables elsewhere. This mode can be used, however, in applications not designed for use with a cartridge. Arcade-type games are one example: the game stands alone, and you are relatively free to use more of page zero for your own use.

OTHER ADDRESSING MODES

We have now discussed 8 of the 13 available addressing modes of the 6502. The remaining five modes cannot be demonstrated using the LDA command, since it uses only these eight modes. We will now discuss the others, using other commands in the instruction set.

ACCUMULATOR MODE

The ROTATE and SHIFT instructions, ROR, ROL, ASL, and LSR, can all use an addressing mode known as **accumulator mode**:

```
ROR A
```

or

```
ASL A
```

This simply means that the rotation or shift is to be performed on the contents of the accumulator rather than the contents of some memory location. Note that these instructions can also operate on memory:

```
ROL $0523
```

IMPLIED MODE

Many of the instructions can use an **implied mode** of addressing, where the addressing mode is obvious from the instruction. For instance, DEX and CLD are both 1-byte instructions whose addressing target is obvious from the instruction itself.

RELATIVE MODE

The BRANCH instructions all use the **relative addressing mode**. That is, one can read the branch as meaning either to branch forward 10 bytes, or to branch backward 4 bytes. The branch is relative to the current position of the program counter.

INDIRECT MODE

The JMP instruction can use the indirect form of addressing. For example, if we set up an address by storing #$53 in location $CD and #$E4 in location $CC, we can jump indirectly to $E453:

```
JMP ($CC)
```

ZERO PAGE, Y MODE

The final form of addressing is called the **zero page, Y** form. This second zero page indexed mode is used only by two instructions, LDX and STX. That is, when the X register is used to load or store a value, it may be indexed with the Y register from a zero page base address. This is virtually identical to the zero page, X mode we have already discussed.

This concludes our review of the addressing modes available using the 6502. In later chapters, we will see how these modes can be used to accomplish useful programming chores, and the benefits of having more than just one or two addressing modes will become obvious.

CHAPTER SIX
ASSEMBLERS FOR THE ATARI

BACKGROUND

When we talk about **assemblers**, generally we are speaking about software packages which allow us to write programs in assembly language and get them to run. Such software packages usually contain three parts: an **editor**, used to actually write the source code programs; an **assembler**, used to convert the source code program into machine language, which will actually run; and a **debugger**, used to find errors and correct them, so that your finished product works the way you intended.

There are currently six assembler packages available for ATARI computers:

1. The Assembler/Editor Cartridge, from ATARI, Inc.
2. ATARI Macro Assembler (AMAC), MEDIT (an editor), and DDT (a debugger), all from the APX, ATARI, Inc.
3. MAC/65, from Optimized Systems Software, Inc., 10379 Lansdale Avenue, Cupertino, California 95014.
4. The SYNASSEMBLER, from Synapse Software, 5327 Jacuzzi, Suite 1, Richmond, California 94804.
5. The Macro Assembler/Text Editor (MAE), from Eastern House Software, 3239 Linda Drive, Winston-Salem, North Carolina 27106.

6. Edit 6502, from LJK Enterprises, P.O. Box 10827, St. Louis, Missouri 63129.

Since it is the most widely owned, although certainly not the most powerful, assembler available for the ATARI computer line, all of the examples in this book will be written using the ATARI Assembler/Editor Cartridge. This chapter will describe the syntax used in the Cartridge, and further explain how each of the other five assemblers compare with it.

It is not the purpose of this book to endorse, either directly or indirectly, any of these products. These assemblers, and particularly the differences between them, are described here to enable you to work with the examples in this book and use the routines for your own, no matter which of the products you have purchased.

THE ATARI ASSEMBLER/EDITOR CARTRIDGE

First, let's discuss syntax. The Assembler/Editor Cartridge requires that every line be prefaced with a line number, as do any BASIC programs you have written. Using the Cartridge, these line numbers must be integers between 0 and 65535. Each line number must be followed by at least one blank space. The fields which are present in a line of an assembly language program are:

line number label mnemonic operand comment

For an example, we'll look at one typical line of a such a program:

```
10 LOOP LDA $0342  ;start by getting hi byte of variable X
```

Let's take one part of this line of assembly language code at a time. The first field, the **label** field, may or may not be present. If it is present, you can tell the assembler to address this line by using its label, in this case, LOOP. Therefore, we could subsequently write another line of code which branched to LOOP, and the assembler would know where we wanted to go. Generally, labels are used only when we know we will later need to reference this line from another portion of the program. As mentioned above, the label field, if present, must have exactly one blank space between the last digit of the line number and the first character of the label. The first character of the label must be a letter from A to Z, and the other characters must be either letters or the digits 0 through 9. The label may be as short as 1 character or as long as (106 minus the number of digits in the line number). Since some of the assemblers for the ATARI limit the number of characters which may be used in the labels, all label names used in the programs in this book will contain six or fewer characters.

The **mnemonic**, often called the **op code**, is the 6502 instruction that we wish the computer to execute at that point in the program. In the example given above, we want the computer to load the accumulator, and the mnemonic for this is LDA, as we learned in Chapter 5. This instruction must appear either with one blank space between itself and the label, if there is a label, or with two blank spaces between the last digit of the line number and the first letter of the mnemonic, if there is no label. For example:

```
10 LABEL LDA $0342    or    10  LDA $0342
```

The reason for this should be apparent: if only one such blank space were left after the line number, the assembler would try to apply the mnemonic as a label, and you'd end up with a label called LDA. The assembler would then try to interpret the operand, $0342, as a 6502 instruction, without any success whatsoever.

The **operand** is the conclusion of the 6502 instruction, and specifies the address or number we would like to operate on. For instance, in this case the operand defines the absolute addressing mode, in which the accumulator is to be loaded with the number stored in memory location $0342. The operand could have been #$24, in which case the addressing mode would have been immediate, and the accumulator would have been loaded with the hexadecimal number $24, instead of a number from someplace in the computer's memory. The operand starts with at least one blank space between its first character and the last character of the mnemonic, although more blank spaces are permitted. In fact, you can tab over to the operand field if you so desire. In this book, we'll use one blank space.

With any mnemonic which uses the accumulator addressing mode, the operand must be the capital letter A to be properly interpreted by the Cartridge. Therefore, an instruction to rotate the contents of the accumulator to the right it must be written like this:

```
130 ROR A ;note the A
```

The **comment** field is the final field of a line of assembly language code, and it should describe the operation being performed in terms of program function. That is, the comment should not describe the operation (that LDA $0342 means to load the accumulator from $0342); rather the comment should remind you what that particular line of code is doing, so that you can go back to it 6 months later and not spend 10 hours wondering what in the Sam Hill that stupid line was for. In our first example above, the comment tells us that we're getting the high byte of a variable we're calling X, from memory location $0342.

Comments can be set off from code in two ways. First, if at least one blank space follows the operand field, anything else following on that line will be interpreted by the assembler as a comment. A second way is to denote an entire line as a comment. As we shall see, this often makes your code much more readable and will be a big help in keeping your sanity. To so designate a line, follow the line number with one space and place a semicolon in the next space. Anything else on that line will be interpreted as a comment at assembly time. Examples of each of these methods are:

```
100 ; This entire line is a comment line
100 LDA $0343 ;This is a comment also
```

For the purposes of this book, all comments, either full-line or not, will be preceded by a semicolon, so if you see a semicolon before some text, you'll know that you're reading a comment.

Now we know the structure of a line of assembly language code and there are a few other conventions that we'll need to know as well.

Directives

Most assemblers have available for the programmer's use a series of instructions which can be interpreted by the assembler, essentially extending the instruction set of the 6502. These are called **directives**, or, sometimes, **pseudo-ops**, since they are used just like op codes but are not part of the 6502 instruction set. The most important of these for the Assembler/Editor Cartridge are described below, with a brief description of each.

One of the most important is the **origin** statement. Since the assembler creates machine language code which will reside in a specific place in memory, we need to tell the assembler where this place is. To do this, we use the origin statement. With the Cartridge, the format of this statement is as follows:

```
10  *= $0600  ;the beginning
```

Note that there are two spaces between the last digit of the line number and the asterisk, no spaces between the asterisk and the equal sign, and one space between the equal sign and the first character of the address. This line tells the assembler that we want our code to begin assembly at hexadecimal address $0600, or page 6. Such an origin statement will usually be the first, or one of the first, statements in our programs. When the assembler sees the *= directive, it assigns the program counter the value of the expression following this directive. It is perfectly feasible to have more than one *= directive in any program if different regions of code are to be assembled in different areas of memory.

Other pseudo-ops include:

.BYTE reserves at least one location in memory for future use. The operand can place information into this space. For instance, the instruction

```
110  .BYTE 34
```

opens one location in memory at the current position of the program counter, and stores the number #$22 (decimal 34) in that location. It is also possible to store a series of bytes using one .BYTE instruction, as shown below:

```
125 .BYTE "HELLO",$9B
```

This will store the hexadecimal numbers $48, $45, $4C, $4C, $4F, and $9B in consecutive locations. These numbers are the ATASCII (**AT**ari **ASCII**) codes for the letters of the word HELLO.

.DBYTE reserves two locations for each value in the operand. This instruction is used for data in which the numbers are larger than 256, and so require 2 bytes to be stored. The number is stored with the high-order byte first, followed by the low-order byte. For example,

```
115 .DBYTE 300
```

stores 2 hexadecimal numbers in consecutive memory locations. The first is $01 and the second is $2C, since 300 decimal equals $012C hexadecimal.

.WORD is identical to the .DBYTE directive, except that the low-order byte is stored first, followed by the high-order byte.

LABEL = is used to assign a value to a label. For instance, if we write a program which requires the frequent use of the address $9F, we can assign this address to a named variable, as follows:

```
112 FREQ = $9F
```

Since the label in the LABEL = directive is a real label, it must begin with exactly one blank space between the last digit of the line number and the first character of the label. Now, whenever we need the address, we can call the label instead; for instance,

```
245 LDA FREQ
```

The assembler now knows to load the accumulator from the address $9F.

.END tells the assembler that it has completed the assembly and that it should stop right there. Obviously, it should be the last line of your program. The Assembler/Editor Cartridge assumes that if there are no further lines of code and no .END directive is included, the program is finished; this makes the .END directive optional, much as the END statement in an ATARI BASIC program is optional.

There are many other directives available for the Cartridge, but we have discussed the most important ones and for the moment they are the only ones we'll discuss. Please refer to the Assembler/Editor manual for a further discussion of all of the pseudo-ops available.

OPERAND FIELD MATH

One further note on the Cartridge is that it supports addition, subtraction, multiplication, and division in the operand field. For instance, if we would like to break up the address of the label LOOP into a high and a low byte, we can write the following section of code:

```
135  LDA #LOOP&255   ;get low byte of LOOP
140  STA DEST        ;store it in DEST
145  LDA #LOOP/256   ;get high byte
150  STA DEST+1      ;and we're done.
```

Line 135 takes the address of LOOP and ANDs it with #$FF, giving us the low-order byte. Line 145 divides this address by 256, giving us the high-order byte of the address. Note that line 150 will store this byte in the address DEST plus 1, or 1 byte higher in memory than DEST.

THE ATARI MACRO ASSEMBLER

We will now discuss the differences between the other available assemblers and the Assembler/Editor Cartridge.

A macro assembler allows you to write, cleverly enough, **macros**, which are generally short segments of assembly language code that you plan to use frequently within a program. An example of a macro might be JMI, which would contain the code to implement a Jump on Minus instruction, which, as we now know, is not present in the 6502 instruction set. Using a macro assembler, we could code this instruction, and then whenever we want to jump on minus, we could use JMI very much like a normal instruction of the 6502 set. At assembly time, the assembler will find the right macro and insert it properly everywhere the JMI instruction occurs.

The ATARI Macro Assembler is unique among the available assemblers in its ability to assemble one single file many times larger than the entire memory space of the ATARI! It does this by reading code from your disk drive, assembling it, and writing the assembled object code back to another disk file. It has another powerful feature — the use of separate files, called SYSTEXT files. These can include all label references, so that such a SYSTEXT file can be constructed once, with all of the equates for the ATARI contained in it. This file can then be used for all programs you'll ever write, without the need for laboriously constructing this table again.

This assembler is also different in one other respect — it uses no line numbers. Lines are simply inserted or deleted in the appropriate order, and your program scrolls through memory. To see the beginning of your program, you scroll the text down until the beginning appears, and vice versa for the end of your code. When using this assembler, you should take care to place the lines in the correct order, or else you'll have trouble at run time.

The label begins in column 1, and the mnemonic is generally tabbed over about eight spaces. The operand and comment fields follow the operand. Labels may be of any length, but only the first six characters are significant; longer labels are used at your own risk. Octal numbers, in addition to binary, decimal, and hexidecimal, are supported, and when used, are prefaced by the @ sign. Strings, such as the HELLO used as a previous example, are enclosed in single, rather than double, quotation marks. The AMAC assembler also supports addition, subtraction, multiplication, and division, as well as a number of logical operations. An address may be broken into its high- and low-order bytes simply by using the words HIGH and LOW, without the need for division as in the example above. Macros are, of course, allowed. For any instructions which utilize the accumulator mode, the letter A should follow the mnemonic. The pseudo-ops are virtually all different from those of the Cartridge. The ATARI Macro Assembler (AMAC) and the Assembler/Editor Cartridge versions of the pseudo-ops are outlined below:

AMAC	Cartridge	Comment
DB	.BYTE	(.BYTE also acceptable for AMAC)
DW	.WORD	(.WORD also acceptable for AMAC)
END	.END	(.END also acceptable for AMAC)
EQU	=	
LOC		sets location counter for assembly
ORG	*=	

MAC/65

MAC/65 is also a macro assembler, with features similar to those already described. This assembler is the only one which tokenizes your source code, just as BASIC does. In addition, it checks the syntax of your line as soon as you hit RETURN after typing it. This feature is not quite as important in assembly language programming as it is in BASIC, since most assembly language errors are not a result of syntax errors, but rather logic errors. However, for the beginning assembly language programmer, this is a nice feature which may eliminate some simple, common errors. Line numbers in the range of 0 to 65535 are required for each line. As described for the Assembler/Editor Cartridge, the line number must be followed by a single space before a label, and by 2 spaces before the mnemonic in lines with no labels. Strings within a program must be enclosed in double quotation marks, as when using the Cartridge. Comments begin at least 1 space beyond the operand field, and need not be prefaced by a semicolon. Comments which take an entire line must be prefaced by either a semicolon or an asterisk. Any instructions using the accumulator mode of addressing require the mnemonic to be followed by the capital letter A, as with the Cartridge. Labels may be up to 127 characters long, with all characters significant.

MAC/65 supports addition, subtraction, multiplication, and division. However, whereas the Cartridge has no operator precedence and simply evaluates a complex arithmetic expression from left to right, this assembler has the usual multiplication > division > addition > subtraction operator precedence. It also uses the > and < symbols to designate the high and low bytes, respectively, of an address.

This assembler can be used with files created by another assembler: an ENTER command will allow such files to be read into the editor. In fact, unnumbered lines such as those produced by AMAC can even be numbered automatically using the ENTER command. Then minor changes in syntax will allow the program to be modified and assembled using MAC/65.

The pseudo-ops discussed above for the Cartridge are used in exactly the same way for MAC/65, and the *= symbol for the origin statement is also used in both assemblers. Macros are supported, and the debugger which comes with MAC/65 is a separate program which must be loaded separately, much as AMAC.

THE SYNASSEMBLER

The SYNASSEMBLER also requires line numbers, which must be followed by a blank space before entering a label. The acceptable line number range is from 0 to 63999. Tab stops are built into the editor to allow easy formatting of lines of code, and in fact, the automatic line-numbering mode requires the use of the tab to print the new line number to the screen. Labels may be up to 32 characters long, and all characters are significant. The accumulator addressing mode does not use the letter A following the mnemonic. Comments require semicolons only when whole lines are used for comments.

The SYNASSEMBLER supports only addition and subtraction. Therefore, you'll need to write code to support other operations, or use calculated numbers rather than complex arithmetical expressions. Operators are included, however, to separate a number into high and low bytes — the # and / symbols, respectively. For example,

```
124 LDA #STOR1 ;indicates low byte of STOR1
128 STA STOR3
132 LDA /STOR1 ;indicates high byte
```

The pseudo-ops of the SYNASSEMBLER are considerably different from those of the Cartridge, as you can see in the following chart:

SYNASSEMBLER	Cartridge	Comment
.AS	.BYTE	for ASCII literals only
.BS	.BYTE	
.DA	.WORD	
.EN	.END	
.EQ	=	
.OR	*=	

The SYNASSEMBLER also has an ENTER command, which will allow you to use it to assemble code produced by one of the other packages discussed in this chapter. One use you might make of this feature is to have the SYNASSEMBLER assemble code originally written using the Cartridge, since the SYNASSEMBLER is from 50 to 100 times faster in assembling code than is the Cartridge. For short programs this difference is relatively insignificant, but for long programs, the time needed to assemble code is a substantial portion of the debugging process. Cutting this time substantially will yield a much more productive editing session.

Finally, strings may be surrounded by any delimiter, so either single or double quotation marks can be used.

THE MACRO ASSEMBLER/TEXT EDITOR (MAE)

MAE is another macro assembler available for the ATARI computers. It requires line numbers in the range of 0 to 9999. Any label in a line must immediately follow the line number *with no intervening space!* From the label on, fields are separated by spaces. Semicolons are required at the beginning of full line comments only; comments at the end of a line need only be separated from the operand by a space. The accumulator mode requires that the letter A follows the mnemonic.

MAE supports only addition and subtraction, but two symbols are included for calculating the high and low bytes of a number — #H and #L, respectively. Labels may be up to 31 characters long, with each character significant.

The pseudo-ops supported by MAE, with their Cartridge equivalents, are charted below:

MAE	Cartridge
.BA	*=
.BY	.BYTE
.EN	.END

Single quotation marks are required around strings. One important difference between MAE and all of the other assemblers is that with MAE, any reference to zero page addressing must begin with an asterisk. For instance, if STOR1 and STOR2 are both defined to reside on page zero, then the code to load the accumulator from STOR1 and then to store this value in STOR2 would need to be written like this:

```
105  LDA *STOR1
110  STA *STOR2
```

EDIT 6502

This assembler does not require line numbers for the assembler code. It supports addition, subtraction, multiplication, and division, with no precedence; complex arithmetic expressions are evaluated from left to right. The program counter can be referenced by the use of the asterisk. Strings can be surrounded by either single or double quotation marks. If single marks are used, the high bit of each byte is cleared (set equal to zero), and if double marks are used, the high bit of each byte is set (made equal to 1). The accumulator addressing mode does not use the letter A, just the mnemonic, as here:

```
LDA $4235
ASL
```

The > symbol can be used to generate the high byte of a number, and the symbol will generate the low byte.

Edit 6502 uses a number of pseudo-ops which are different from those of the ATARI Assembler/Editor Cartridge, as described below:

Edit 6502	**Cartridge**
EQU	=
ORG	*=
DFB	.BYTE
DFW	.WORD
END	.END

Now that we've explored some of the differences you'll need to know about to use any of these assemblers for the ATARI computers, we can begin to write some useful assembly language programs. The next chapter explores some subroutines which can be used from BASIC to substantially speed up a program.

CHAPTER SEVEN
MACHINE LANGUAGE SUBROUTINES FOR USE WITH ATARI BASIC

LOCATING MACHINE LANGUAGE PROGRAMS IN MEMORY

When we begin writing subroutines, we must decide where we want to locate them. There are two types of machine language programs, **relocatable** and **fixed**. Fixed programs are those which use specific addresses within the program; these addresses cannot change. For instance, suppose our program contained these lines:

```
30  *= $600
45  LDA ADDR1
50  BNE NOZERO
55  JMP ZERO
60 NOZERO RTS
70 ZERO SBC #1
80  RTS
90 ADDR1 .BYTE 4
```

In this excerpt, we use several references to addresses within the program which are fixed: they cannot change without completely messing up the program. These are more easily seen after we use the assembler to assemble this program, producing output which looks like this:

```
ADDR  ML     LN LABEL   OP    OPRND
0000         30         *=    $600
0600 AD0C06 45          LDA   ADDR1
0603 D003   50          BNE   NOZERO
0605 4C0906 55          JMP   ZERO
0608 60     60 NOZERO   RTS
0609 E901   70 ZERO     SBC   #1
060B 60     80          RTS
060C 04     90 ADDR1    .BYTE 4
```

This output is nicely formatted in columns. The first column lists the hexadecimal addresses at which the machine language instructions translated from the mnemonics are located. The second column lists the machine language code which results from that translation. For instance, the instruction RTS in line 80 generated the machine language code 60, which was located in memory location $060B. The third column lists the line numbers of the assembly language program. The fourth column contains any labels which were present in the original program, and the fifth column contains the mnemonics of the program. The sixth column contains the operand. In this example, there is no seventh column, which would have been present if the original program had contained any comments.

To return to the problem of the fixed addresses discussed above, the first of the problem addresses, ADDR1, can be found in lines 45 and 90. Let's look for a moment at the machine language code which the assembler produced for line 45. Three bytes were produced; AD, 0C, and 06. AD is the machine language code for the absolute addressing mode of the LDA instruction. Since we know that the absolute addressing mode of the LDA instruction requires 3 bytes, we know why 3 bytes were produced by the assembler. The second and third bytes, 0C and 06, make up the address from which to load the accumulator, in the standard 6502 order of least significant byte-most significant byte. Therefore, the address from which to load the accumulator is $060C, which is the address of the line containing the label ADDR1. When we wrote LDA ADDR1, the assembler translated this to mean LDA $060C, since that was the address assigned to ADDR1.

Now we can understand why any attempt to run this program somewhere else in memory is doomed to failure. When line 45 is executed, the microprocessor will look at the original address, $060C, in order to load the accumulator; it expects to find ADDR1 there, since this was the location which was assigned for ADDR1 at the time of assembly. However, the logic of the program was established to perform certain functions based on the value stored in $060C only if $060C was equal to ADDR1. If we move the routine in memory, ADDR1 will be somewhere other than at $060C, and the logic of the program will no longer be valid.

The second fixed address referred to in this program is in line 55. Every JMP instruction has as its destination a fixed address. We can see this by examining the machine language code generated for line 55: 4C, 09, 06. The byte 4C is the machine language code for an absolute JMP instruction, a 3-byte instruction. The next 2 bytes are the address to which to jump, $0609 (remember, least significant byte first). We can now see that when line 55 is executed, the program will jump to $0609, regardless of where in memory we may have moved this program. However, if we do move this program elsewhere in memory, the instruction we intended to have executed at $0609 (the SBC #1 in line 70) will no longer be there. In fact, there will probably be no valid instruction at all at $0609, so the program will crash.

Look for a moment at line 50. Remember that all branch instructions use the relative form of addressing. If we look at the machine language code for this instruction, we'll find D0, 03. D0 is the machine language code meaning to branch on not equal to zero, but that 3 doesn't look like an address. It's not. It simply tells the 6502 to branch forward 3 bytes in memory from the current location of the program counter. When line 50 is executed, the program counter is pointing to the start of the next line. In this case, it points to the 4C of the JMP instruction in line 55. Moving it forward 3 bytes will then point it to 60, the RTS instruction in line 60. That is, when the BNE NOZERO is assembled and executed, this instruction tells the 6502 to branch forward 3 bytes, past the JMP instruction to the address NOZERO ($0608). Since the branch instruction simply says "Branch forward three bytes," rather than "Branch forward to $0608," it can be located anywhere in memory and the branch will still wind up at NOZERO, regardless of where in memory NOZERO is. NOZERO will always be 3 bytes ahead of the branch instruction, so all will be well.

> **PLEASE NOTE!!!** This teaches us an important lesson: branches can be included in relocatable code, but JMPs and specific addresses within the program cannot be.

Why is so much written about relocatable code? For one very simple reason: if code is not relocatable, then we need to find some safe place in the computer's memory to store it. This may not always be easy, since we're sharing the computer with BASIC, and we can't always be sure what locations BASIC will be using. If our code is relocatable, we can put it anywhere. But where?

Let's for a moment review how BASIC handles strings. When you want to use a string in ATARI BASIC, it must be dimensioned. When this is done, the computer reserves space for the string in memory. If for some reason it needs part of that space, it simply moves the string somewhere else, but BASIC is then responsible for remembering where the string is, and is also responsible for protecting its space. Aha! Now we're out of the situation where we have to protect some area of memory from BASIC, and into one in which BASIC does the allocating and protecting for us! We can then store our machine language program as a string in BASIC and access it by using the USR(ADR(ourstring$)) form of command.

To be fair, there will usually be room for a short routine on page 6. Remember that page 6 is guaranteed to always be kept free for the programmer's use. Well, almost always. You should be aware that there is a not infrequent condition under which page 6 may not be safe. As was mentioned in Chapter 3, the space from $580 to $5FF (the top half of page 5) is used as a buffer (a place for temporarily storing information) by your ATARI. If you're entering information from the keyboard, this input buffer may overflow into the bottom of page 6. This overflow will then overwrite anything stored between $600 and $6FF, depending on how much overflow there is. For the purposes of this book, we will assume that such overflow will not occur, and programs which cannot be written to be relocatable will generally have their origin at $600. If they don't work in some specific application you may have, check to be sure that you're not overflowing the buffer from page 5.

Other places to locate non-relocatable programs are up high in memory, or below LOMEM. Both places are generally safe from interference with BASIC if care is taken in their use. Additionally, if you have an application which will *never* use the tape recorder, you may use the tape buffer, located between $480 and $4FF, for program location. Very small routines may also be placed at the low end of the stack, from $100 to about $160, since only rare applications will ever use the stack to this depth. However, this is extremely risky, and no guarantees about safe performance can be given for programs using this space.

While we're on the subject of tape recorders, one final note about the organization of this book. All programs are written assuming the presence of a disk drive and a resident DOS. If you are using a tape-based system, please refer to your assembler manual for instructions on how to perform certain operations. For instance, loading machine language files from a disk drive may use the L option of DOS, whereas the same operation using a tape recorder will probably use some form of the LOAD or BLOAD commands, depending on which assembler you are using.

A SIMPLE EXAMPLE SUBROUTINE TO CLEAR MEMORY

Let's begin to build our library of subroutines with a very simple example. Remember, if you don't want to type all of the programs, they are available on disk from MMG Micro Software.

In BASIC, we frequently need to clear an area of memory to zeros. This occurs, for instance, when using player-missile graphics or when using memory as a scratchpad or even just when a screen or drawing needs to be cleared. Remember that if we want to store data near the top of memory, the display list and display memory must be relocated below this area of memory, or else this routine will wipe out the picture on our TV screen. This relocation can be accomplished very easily, using the following code in BASIC:

```
10 ORIG=PEEK(106):REM Save original top of memory
20 POKE 106,ORIG-8:REM Lower the top of memory by 8 pages
30 GRAPHICS 0:REM Reset the display list and screen memory
40 POKE 106,ORIG:REM Restore the top of memory as before
```

We can, of course, perform the memory clearing operation in BASIC. If we need to clear the top 8 pages of memory to all zeros, we can do so with the following program:

```
10 TOP=PEEK(106):REM Find the top of memory
20 START=(TOP-8)*256:REM Calculate where to start the clear
30 FOR I=START TO START+2048:REM Area to be cleared
40 POKE I,0:REM Clear each location
50 NEXT I:REM All finished
```

This program works just the way we want it to, but takes approximately 13.5 seconds to execute. If we need to perform this operation several times during the course of a program, or if we have a program which cannot afford the 13.5 seconds required to do this in BASIC, then we have a good candidate for a machine language subroutine.

First we'll need to think about where we'll locate our subroutine. Page 6 is as good a place as any. Then we'll need to know how to find the top of memory, so we'll know what part of memory we want to clear. There is a memory location which always keeps track of where the top of memory, in pages, is in an ATARI, location 106, so that part is easy. Finally, this type of program is usually a good candidate for indirect, Y addressing, so we'll need two page zero locations to hold our indirect address. Let's write what we have so far, in assembly language:

```
100 ; ***************************
110 ;set up initial conditions
120 ; ***************************
```

```
130  *= $600 ;we have to assemble it somewhere
140 TOP = 106 ;here's where we find the top stored
150 CURPAG = $CD ;where we store current page being cleared
```

Note that there are only four page zero memory locations, $CC to $CF, which are secure from being changed by both BASIC and the Assembler/Editor cartridge: we'll use two of them, $CC and $CD, for this program. It is possible to find other page zero locations safe from the cartridge, but these are guaranteed by ATARI always to be safe, so we'll use these.

Now let's think about what we'd like the routine to do. First we must remember the PLA required to pull the number of parameters, passed by BASIC in the USR call we'll write, off the stack. After we've found the present top of memory, we'll need to start 8 pages below that, clearing all of memory to zero. The code to find where in memory to start clearing is fairly straighforward, as shown below:

```
160 ; ***************************
170 ;begin with calculations
180 ; ***************************
190  PLA     ;remove # of parameters from stack
200  LDA TOP ;find the top
210  SEC     ;get ready for subtraction
220  SBC #8  ;find first page to clear
230  STA CURPAG ;we'll need it
240  LDA #0 ;to insert it in memory later
250  STA CURPAG-1 ;the low byte of a page # is always zero
```

Now we've set up the indirect address of the first place in memory to clear, and we've stored it in CURPAG-1 (low byte) and CURPAG (high byte). We've also loaded the accumulator with zero, so we're all set to store a zero in each memory location we need to clear.

Next, we need a counter to keep track of how many memory locations have been cleared on each page. If we use the Y register for this, it can act both as a counter and as an offset for the addressing system we're using. All we have to do is set the counter, store the zero in the accumulator into the first location, and decrement the Y register by 1, looping back to perform the store again. Since we began with the counter at zero and we are decrementing by 1 as we clear each location, as long as the Y register has not yet reached zero again, we still have more to clear on this page, since there are 256 locations per page of memory. Let's see what the code will look like:

```
260  LDY #0 ;for use as a counter
270 ; ***************************
280 ;now we'll enter the clearing loop
290 ; ***************************
300 LOOP STA (CURPAG-1),Y;the first byte is cleared
310  DEY ;lower the counter
320  BNE LOOP ;if >zero, page is not yet finished
```

Now that was pretty simple. We stored the zero that was in the accumulator into the address pointed at by the indirect address CURPAG-1, CURPAG (low byte, high byte) offset by Y, which was zero the first time through the loop. Then we decreased the Y register by 1, and looped back to store a zero in the memory location pointed to by the same indirect address, but this time offset by 255, so we've cleared the top byte of the page. Next time through, Y equals 254, so we clear the next-lower byte of memory, and so on, until when Y equals zero the whole page is cleared and our counter is back to zero, ready for the next page.

All right, now we've cleared 1 page. How do we get it to clear all of the other 7 pages? Remember that the indirect address we set up on page zero has 1 byte for the low byte of the indirect address pointing to the page to be cleared, and one byte for the high byte of the address. If we simply increase the high byte by 1, this indirect address will point at the next-higher page, like this:

```
330  INC CURPAG ;to move on to next page
```

It really couldn't be much easier than that, could it? Now all we need to do is find out when we're done.

In this case, we know that we're done when the page we're clearing is higher than the top of memory. It is fairly easy to determine if this condition is true, as follows:

```
340  LDA CURPAG ;need to see if we're done
350  CMP TOP ;is CURPAG>TOP?
360  BEQ LOOP ;no, last page coming up!
370  BCC LOOP ;no, keep clearing
380  RTS ;go back to BASIC
```

If CURPAG is equal to TOP, remember that we've still got that last page to clear. Only if CURPAG is greater than TOP have we finished.

Now that we've written our program, we need to convert it from assembly language to machine language. To do this, we use the assembler part of the cartridge, which can be accessed simply by typing ASM followed by a RETURN. This will start the assembly process, and after a short pause, the following information will appear on your screen:

```
ADDR   ML    LN   LABEL  OP     OPRND        COMMENT
---------------------------------------------------------------------
             0100 ;*********************************
             0110 ;set up initial conditions
             0120 ;*********************************
0000         0130        *=     $600         ;place to assemble it
006A         0140 TOP    =      106          ;where the top is stored
00CD         0150 CURPAG =      $CD          ;to store page being
                                               cleared
             0160 ;*********************************
             0170 ;begin with calculations
             0180 ;*********************************
0600 68      0190        PLA                 ;# of parameters off stack
0601 A56A    0200        LDA    TOP          ;find the top
0603 38      0210        SEC                 ;get ready for subtraction
0604 E908    0220        SBC    #8           ;find first page to clear
0606 85CD    0230        STA    CURPAG       ;we'll need it
0608 A900    0240        LDA    #0           ;to insert it in memory later
060A 85CC    0250        STA    CURPAG-1     ;low byte of page # is zero
060C A000    0260        LDY    #0           ;for use as a counter
             0270 ;*********************************
             0280 ;now we'll enter the clearing loop
             0290 ;*********************************
060E 91CC    0300 LOOP   STA    (CURPAG-1),Y;the first byte is cleared
0610 88      0310        DEY                 ;lower the counter
0611 D0FB    0320        BNE    LOOP         ;if >zero, page not done yet
```

```
0613 E6CD   0330        INC  CURPAG     ;let's move on to next page
0615 A5CD   0340        LDA  CURPAG     ;need to see if we're done
0617 C56A   0350        CMP  TOP        ;is CURPAG>TOP?
0619 F0F3   0360        BEQ  LOOP       ;no, last page coming up!
061B 90F1   0370        BCC  LOOP       ;no, keep clearing
061D 60     0380        RTS             ;go back to BASIC
```

Now we have assembled the program and stored it in memory. The next task is to store it onto our disk so we can use it in our BASIC program. This can be done in either of two ways. The first is directly from the cartridge, using the SAVE command as follows:

```
SAVE#D:PROGRAM<0600,061F
```

This command creates a file on disk called PROGRAM, and stores all of the contents of memory from $0600 to $061F in that file. Note that we've stored a few extra bytes — generally a good idea.

The second way to store this information is to go to DOS and save memory using the K option. This can be done as follows:

```
PROGRAM,0600,061F
```

Either method of storing the results of the assembly will be satisfactory.

Now you can switch cartridges, and replace the Assembler/ Editor with the BASIC cartridge. After booting up the computer, type DOS, and when the DOS menu appears, use the L option to load the file called PROGRAM that we just created. Then type B to go back to BASIC.

Our program now resides on page 6, and we can access it if we like. However, the next step should be to put it into a form that doesn't require the use of DOS for loading. We could simply write one line of BASIC code in the direct mode to pull this information from page 6; for example,

```
FOR I=1 TO 30:?PEEK(1535+I);" ";:NEXT I
```

However, since we're using a computer, why not write a general-purpose program that will pull the data out of memory and set it up in a form which we can convert easily to DATA statements in a BASIC program? Such a program is given below:

```
10 FOR J=1 TO 30 STEP 10:REM Length of data in memory
20 FOR I=J TO J+9:REM We'll get DATA statements 10 bytes long
30 PRINT PEEK(I+1535);",";:REM Print the data to the screen
40 NEXT I:REM Finish the line
50 PRINT:PRINT:REM Leave blank lines for easy working
60 NEXT J:REM All done!
```

If we now type this program in and RUN it, our screen will show the following:

```
104,165,106,56,233,8,133,205,169,0,
133,204,160,0,145,204,136,208,251,230,
205,165,205,197,106,240,243,144,241,96,
```

It's now a simple matter to move the cursor up to these lines, remove the trailing commas, and convert them to the following:

```
10000 DATA 104,165,106,56,233,8,133,205,169,0
10010 DATA 133,204,160,0,145,204,136,208,251,230
10020 DATA 205,165,205,197,106,240,243,144,241,96
```

Now we can erase lines 10 to 60, so that the program in memory consists of just lines 10000 to 10020. At this point, we should save the program to disk, so we don't have to go through this whole procedure again if the power fails. We can incorporate this routine into a short BASIC program to test it, as follows:

```
10 FOR I=1 TO 30:REM Number of bytes
20 READ A:REM Get each byte
30 POKE 1535+I,A:REM POKE byte in correct location
40 NEXT I:REM Finish POKEing data
50 ORIG=PEEK(106):REM Now relocate display list, as above
60 POKE 106,ORIG-8
70 GRAPHICS 0
80 POKE 106,ORIG:REM Restore top of memory
90 POKE 20,0:REM Set timer to zero
100 X=USR(1536):REM Call our machine language routine
```

```
110 ? PEEK(20)/60:REM How many seconds did it take?
120 END:REM Separate DATA from program
10000 DATA 104,165,106,56,233,8,133,205,169,0
10010 DATA 133,204,160,0,145,205,136,208,251,230
10020 DATA 205,165,205,197,106,240,243,144,241,96
```

Line 90 first sets the internal real-time clock to zero, and then line 110 reads the time in jiffies (sixtieths of a second). This will measure the elapsed time the USR call, our machine language routine, took to clear 8 pages of memory. It takes 0.0333 seconds, so this machine language routine, which seems so long and time-consuming, is over 400 times faster than the BASIC program that did the same job. Worth the effort, wasn't it?

Of course, all that time spent programming this routine was not wasted, since we've now got a routine which we can use whenever we need to clear the top of memory, such as for player-missile graphics.

The program we wrote has one drawback: the code resides on page 6 and therefore cannot be used in a program which needs page 6 for its own use. Now here's where the relocatable nature of the code comes in. Let's look again at the assembly language program we wrote. Note that we didn't use any jumps, nor did we make reference to any address within the program, except in branch instructions. This program is not tied to any specific memory locations; it can reside anywhere in memory and still work. Let's take advantage of that and turn the program into a string. This process is fairly simple. All we need to do is add a line 5 and change lines 30 and 100, as follows:

```
5 DIM CLEAR$(30):REM Set up the string
30 CLEAR$(I,I)=CHR$(A):REM Insert byte into string
100 X=USR(ADR(CLEAR$)):REM New location of the clearing routine
```

So now we have a relocatable routine to clear memory, which will be far more versatile than the one tied to specific locations. In fact, if the string which we create contains no control characters, we can simply produce a 1-line subroutine which will contain all of the information we need. We can do this by running the program and then printing CLEAR$ to the screen. We can then move the cursor up to the string of machine language and convert it to a single line of BASIC, as follows:

```
20000 E=USR(ADR("h%j8i/qM)♥L♥rM/P♣fM%MEjps4q♦")):RETURN
```

It can't be any simpler than this! Now whenever we need to clear the top of memory, we can just include this line of code, and access the subroutine to clear the memory.

Other, more sophisticated routines exist for producing BASIC programs once you have created your machine language routine, or you can write such programs yourself. One very nice routine for producing such subroutines as strings was published in the September, 1983, issue of **ANTIC** magazine, by Jerry White. This routine reads the machine language data directly from the disk and writes the BASIC code back to disk. This avoids one problem with printing such strings to the screen; if the string contains a non-printing character, a fair amount of work is required to be sure the string is correct. One way to check your routines quite easily for such problems is to simply count the number of characters printed to the screen when you print your string, and compare that number to the number of bytes contained in your machine language routine. If the numbers differ, you'd better find out which character has been omitted and insert it in the appropriate place using this key sequence:

```
ESC  CTRL-key
```

which will allow you to print normally nonprinting characters. Let's take a simple example of this. Suppose you have written a machine language routine for some purpose, and when you print the string containing this routine to the screen, it is 1 byte shorter than it should be. Furthermore, you hear a bell sound every time you print this string. In reviewing your DATA statements, you find that the 15th byte of your machine language routine is 253. When you attempt to print the character corresponding to ATASCII 253, the bell will sound, since this is the code for the keyboard buzzer, but the character will not be printed to the screen. To solve this problem, print the string to the screen and then position the cursor over the 15th byte of the string. Press the CTRL key and the INSERT key simultaneously, and from the 15th character on the string will move 1 position to the right, leaving space for the missing character. Now press the ESC key, and next simultaneously depress the CTRL key and the 2 key, and the correct character will be inserted in the 15th byte of your string. A line number and the other information required, as shown above, may then be added, and you'll have your routine on a single line.

For short, single routines, the easiest way around this problem is not to use strings in single lines, but rather to insert the characters in a string using DATA statements, as already demonstrated. However, where single-line strings are desirable — for example, where space is at a premium — the more cautious the programmer, the better the results will be.

SUBROUTINE TO RELOCATE THE CHARACTER SET

One of the very nice features of the ATARI computers is the ease with which the standard character set (normally the uppercase, lowercase, and inverse letters, numbers, and symbols we use every day) can be altered for any purpose we desire. For instance, one of ATARI's most popular games, SPACE INVADERS, was programmed using redefined characters for the attacking invaders. These are then simply printed to the screen in the appropriate place. By printing them all 1 position further right each loop of the game, they appear to march across the screen, in their ominous fashion.

As we know, however, the normal ATARI character set resides in ROM, beginning at location 57344 ($E000 hexadecimal). In order to alter any of the standard characters, we need to move the character set to RAM, where we can get at it. This can, of course, be done in BASIC. A very simple BASIC program to accomplish this is given below:

```
10 ORIGINAL=57344:REM Where character set is in ROM
20 ORIG=PEEK(106):REM Where top of RAM is located
30 CHSET=(ORIG-4)*256:REM Where relocated set will be
```

```
40 POKE 106,ORIG-8:REM We'll make room for it
50 GRAPHICS 0:REM Set up new display list
60 FOR I=0 TO 1023:REM Now we'll transfer the whole set
70 POKE CHSET+I,PEEK(ORIGINAL+I)
80 NEXT I
90 END :REM That's it
```

This program reserves 8 pages of memory near the top of RAM, much like the previous example did. There is no need to clear this area to all zeros in this case, however, since we fill up 4 of the pages with the character set from ROM. The loop from lines 60 to 80 actually accomplishes the transfer of the character set, which is 1024 bytes long (8 bytes per character times 128 characters). The program works fine, and if we don't mind spending 14.7 seconds to accomplish this transfer, we don't need assembly language at all.

If we'd like to go faster, however, we'll need a machine language subroutine to accomplish the transfer. The subroutine we wrote to clear an area of memory to all zeros contains the techniques we will use in such a program. However, we'll need to add two new features. The first will allow BASIC to pass to our subroutine the location at which we would like our character set to reside in RAM, by using the parameter passing discussed in Appendix 1. The second will store different values in each memory location, rather than storing the same character in each location. To do this, we'll need two indirect addresses set up on page zero. In addition, in this routine we will employ the more usual nomenclature, defining our label as being the lower of the two zero page locations and referring to the higher location as label + 1, rather than defining the higher and referring to the lower location as label − 1. Both methods are presented, to demonstrate the flexibility of programming in assembly language. Now, let's begin with the setup:

```
            0100 ;******************************
            0110 ;set up initial conditions
            0120 ;******************************
0000        0130         *=     $600
00CC        0140 FROM    =      $CC
00CE        0150 TO      =      $CE
```

From this point on, we'll use the output from the assembler for all of the programs shown. To type these programs for yourself, just type the line number, label (if present), mnemonic, operand, and comments, and assemble it for yourself. When displayed on your screen, the output of your assembler should look like the output given here. By presenting the programs this way, we can refer to the machine language code generated by the assembler as well as to the assembly language code we write.

As you can see, we have now reserved two different areas of page zero for our indirect addresses. We have defined the lower of each pair of bytes, $CC and $CE, so that the indirect address for the place from which we will get the character set will be stored in $CC and $CD, and the indirect address for the place to which we will move the character set will be stored in $CE and $CF. We have cleverly named these locations FROM and TO. For both sets of locations, the low byte of the indirect address will be stored in the lower of the two locations, and the high byte will be stored in the higher, using typical 6502 convention.

Now that we've reserved space for the indirect addresses, the next task is to correctly fill them with the addresses we need. Remember that we're going to pass the TO address from BASIC, but the FROM address is fixed at $E000 by the operating system. Let's see how this part of the program looks.

```
          0160 ;******************************
          0170 ;initialize and set up indirect addresses
          0180 ;******************************
0600 68   0190  PLA           ;remove # of parameters from stack
0601 68   0200  PLA           ;get high byte of destination
0602 85CF 0210  STA  TO+1     ;store it in high byte of TO
0604 68   0220  PLA           ;get low byte of destination
0605 85CE 0230  STA  TO       ;store it in low byte of TO
0607 A900 0240  LDA  #0       ;even page boundary LSB = 0
0609 85CC 0250  STA  FROM     ;low byte of indirect address
060B A9E0 0260  LDA  #$E0     ;page of character set in ROM
060D 85CD 0270  STA  FROM+1   ;completes indirect addresses
```

Let's discuss lines 190 to 230 for a moment. Line 190 is our old friend, used for pulling the number of parameters passed by BASIC off the stack, to keep the stack in order. Note that both lines

200 and 220 are PLA instructions. This is the method used when passing parameters from BASIC. The number to be passed is broken up by BASIC into high and low bytes and is placed on the stack low byte first, then high byte. Therefore, the first number we pull off the stack is the one on the top, the high byte. We store that appropriately in TO + 1, the high byte of the indirect address we have set up on page zero. Similarly, we store the low byte passed from BASIC in TO, and we have completed setting up the first of the two indirect addresses we will need.

Now we just have to do the easy part. We know that any page boundary has an address with the low byte equal to 0, so we can store a zero in FROM with no difficulty. We know the high byte is $E0, and don't forget the # sign, to let the assembler know that we want to store the number $E0 into FROM + 1, and not whatever number is in memory location $E0.

Now all we need to do is write the loop which will accomplish the transfer for us. We know that we need to transfer 1024 bytes, 4 pages of information, so we'll need a counter to keep track of how far we've progressed. For this purpose, we'll use the X register. We'll also need a counter to keep track of where we are on each page we're tranferring, and for this, we'll use the Y register. Let's see the rest of the program to accomplish this transfer:

```
               0280 ;*******************************
               0290 ;now let's transfer the whole set
               0300 ;*******************************
060F A204      0310           LDX  #4         ;4 pages in the character set
0611 A000      0320           LDY  #0         ;initialize counter
0613 B1CC      0330 LOOP      LDA  (FROM),Y   ;get a byte
0615 91CE      0340           STA  (TO),Y     ;and relocate it
0617 88        0350           DEY             ;is page finished?
0618 D0F9      0360           BNE  LOOP       ;no - keep relocating
061A E6CD      0370           INC  FROM+1     ;yes-high byte
061C E6CF      0380           INC  TO+1       ;high byte-for next page
061E CA        0390           DEX             ;have we done all 4 pages?
061F D0F2      0400           BNE  LOOP       ;no - keep going
0621 60        0410           RTS             ;yes, so return to BASIC
```

There are only two differences between this part of the program and the corresponding part of the previous program we wrote. The first is the use of the X register to determine when we are done. Line 310 sets the X register for the number of pages to be transferred. Lines 390 and 400 determine if we have finished, by decrementing the X register and looping back to continue the transfer if the value of the X register has not yet reached zero.

The second difference, of course, is that we're not going to store the same value in every location, so we need to load the accumulator using the same technique we use to store it, indexing the zero page indirect location with the Y register. Note that when Y equals 1, we'll load from the second location of the ROM character set in line 330 and store it in the second location of the RAM set in line 340, and so on. Remember that lines 370 and 380 raise both indirect addresses by 1 page, since at that point in the program, we will have finished a page, and we'll be ready to begin another.

All that remains is to convert this program into a machine language subroutine for BASIC. With the same technique we used for the first program discussed, we save the machine language code, put BASIC in, load our code back again, and produce DATA statements by PEEKing the values stored from 1536 to 1569. These DATA statements can then be used in a program such as the one given below:

```
10 GOSUB 20000:REM Set up machine language routine
20 ORIG=PEEK(106):REM Top of RAM
30 CHSET=(ORIG-4)*256:REM Place for relocated character set
40 POKE 106,ORIG-8:REM Make room for it
50 GRAPHICS 0:REM Set up new display list
60 POKE 20,0:REM Set timer
70 X=USR(ADR(RELOCATE$),CHSET):REM Relocate the whole set
80 ? PEEK(20)/60:REM How long did it take?
90 END :REM It took 0.03 seconds
20000 DIM RELOCATE$(34):REM Set it up as a string
20010 FOR I=1 TO 34:REM Set up the string
20020 READ A:REM Get a byte
20030 RELOCATE$(I,I)=CHR$(A):REM Stuff it into the string
20040 NEXT I:REM Repeat until string is done
20050 RETURN :REM All done, go back
20060 DATA 104,104,133,207,104,133,206,169,0,133
```

```
20070 DATA 204,169,224,133,205,162,4,160,0,177
20080 DATA 204,145,206,136,208,249,230,205,230,207
20090 DATA 202,208,242,96
```

The subroutine from line 20000 to line 20070 puts each byte of the machine language routine into its appropriate place in a string which we have called RELOCATE$. To access this routine, we use line 70, which passes the parameter CHSET to our machine language routine. Remember that CHSET, defined in line 30, is the address at which we would like to locate the character set in RAM. This program executes almost 500 times faster than the all-BASIC program described above, again demonstrating the speed of machine language routines.

SUBROUTINE TO TRANSFER ANY AREA OF MEMORY

With a few minor modifications to the program we just wrote, we can make it much more versatile. Let's write it in such a way as to allow the transfer of any area of memory to any other area. Looking at the program above, we see that only two parts of the code need to change. The first is the absolute address of FROM, which is set at 57344, and the second is the number of pages stored in the X register, which is set at 4. If we could use variables here instead of constants, our routine would be far more versatile. It's easy to convert the routine in this way; let's just pass the FROM address and the number of pages to transfer as parameters from BASIC. Here is the complete assembly language program for this subroutine:

```
            0100 ;******************************
            0110 ;set up initial conditions
            0120 ;******************************
0000        0130        *=    $600
00CC        0140 FROM   =     $CC
00CE        0150 TO     =     $CE
            0160 ;******************************
```

```
          0170 ;initialize and set up indirect addresses
          0180 ;******************************
0600 68   0190        PLA              ;pull # of parameters off stack
0601 68   0200        PLA              ;get high byte of source
0602 85CD 0210        STA  FROM+1      ;store it in high byte of FROM
0604 68   0220        PLA              ;get low byte of source
0605 85CC 0230        STA  FROM        ;store it in low byte of FROM
0607 68   0240        PLA              ;get high byte of destination
0608 85CF 0250        STA  TO+1        ;store it in high byte of TO
060A 68   0260        PLA              ;get low byte of destination
060B 85CE 0270        STA  TO          ;store it in low byte of TO
060D 68   0280        PLA              ;no high byte exists (=0)
060E 68   0290        PLA              ;get low byte - number of pages
060F AA   0300        TAX              ;put # of pages in X register
          0310 ;******************************
          0320 ;now let's transfer everything
          0330 ;******************************
0610 A000 0340        LDY  #0          ;initialize counter
0612 B1CC 0350 LOOP   LDA  (FROM),Y    ;get a byte
0614 91CE 0360        STA  (TO),Y      ;and relocate it
0616 88   0370        DEY              ;is page finished?
0617 D0F9 0380        BNE  LOOP        ;no - keep relocating
0619 E6CD 0390        INC  FROM+1      ;yes-high byte
061B E6CF 0400        INC  TO+1        ;high byte-now for next page
061D CA   0410        DEX              ;have we done all pages?
061E D0F2 0420        BNE  LOOP        ;no - keep going
0620 60   0430        RTS              ;yes, so return to BASIC
```

We have now set up the routine to obtain first the FROM address in two bytes from the stack, and then the TO address in the same way. Finally, we remove from the stack the number of pages to be transferred. Note that there are only 256 pages of memory in an ATARI, so there can never be a high byte to the number of pages parameter. The low byte is pulled from the stack and transferred to the X register to set up the counter for the number of pages to be transferred.

With the exception of these few changes, the program is identical to our program for transferring the character set from ROM to RAM. In fact, this new routine will accomplish the same goal if we so desire. A BASIC program using this new routine to transfer the character set is given below:

```
10 GOSUB 20000:REM Set up machine language routine
20 ORIG=PEEK(106):REM Top of RAM
30 CHSET=(ORIG-4)*256:REM Place for relocated character set
40 POKE 106,ORIG-8:REM Make room for it
50 GRAPHICS 0:REM Set up new display list
60 X=USR(ADR(TRANSFER$),57344,CHSET,4):REM Transfer the whole set
70 END
20000 DIM TRANSFER$(33):REM Set it up as a string
20010 FOR I=1 TO 33:REM Set up the string
20020 READ A:REM Get a byte
20030 TRANSFER$(I,I)=CHR$(A):REM Stuff it into the string
20040 NEXT I:REM Repeat until string is done
20050 RETURN :REM All done, go back
20060 DATA 104,104,133,205,104,133,204,104,133,207
20070 DATA 104,133,206,104,104,170,160,0,177,204
20080 DATA 145,206,136,208,249,230,205,230,207,202
20090 DATA 208,242,96
```

AN EXERCISE FOR THE READER

Using these techniques, it should be fairly simple to write your own routine to fill a given number of pages of memory with a character other than zero. To obtain the maximum benefit from this exercise, *don't* look back at the examples in this chapter, but rather start from scratch, and see how you do.

READING THE JOYSTICK

We are all familiar with the complex code required in BASIC to read the joysticks. Although there are some sophisticated ways of speeding up this process in BASIC, the most common approach used to determine the position of the joystick and change the X and Y coordinates of a player (for example) is as follows in this subroutine for a BASIC program:

```
10000 IF STICK(0)=15 THEN 10050:REM straight up
10010 IF STICK(0)=10 OR STICK(0)=14 OR STICK(0)=6 THEN
Y=Y-1:REM 11,12 or 1 o'clock position-move player up
10020 IF STICK(0)=9 OR STICK(0)=13 OR STICK(0)=5 THEN Y=Y+1:REM
```

```
7,6 or 5 o'clock position-move player down
10030 IF STICK(0)=10 OR STICK(0)=11 OR STICK(0)=9 THEN
X=X-1:REM 10,9 or 8 o'clock position-move player left
10040 IF STICK(0)=6 OR STICK(0)=7 OR STICK(0)=5 THEN X=X+1:REM
2,3 or 4 o'clock position-move player right
10050 RETURN:REM no other possibilities
```

There are several ways of improving the speed of such a routine by improved programming, as you already know. This routine is included here for simplicity; it is easy to follow its logic. In any case, even excellent programming will not make this type of routine the winner in a speed contest. Let's see if we can speed it up significantly by using assembly language.

We'll assume for the purpose of this example that the joystick routine we shall write will be the only way the player can move, and further, that we will be moving only one player. Since the player can move in only two dimensions, we need only remember two coordinates, the X and Y positions of the player. Because of the way we will be moving the player, we'll need only 1 byte of storage for the X position, but we'll need 2 bytes, for an indirect address, for the Y location. The routine needed for reading the joystick is very straightforward and is given below:

```
0100 ; ******************************
0110 ;initialize locations
0120 ; ******************************
0130  *= $600 ;safe place for routine
0140 YLOC = $CC ;indirect addr. for Y
0150 XLOC = $CE ;to remember X position
0160 STICK = $D300 ;hardware STICK(0) location
0180 ; ******************************
0190 ;now read the joystick #1
0200 ; ******************************
0210  PLA ;keep the stack neat
0220  LDA STICK ;get joystick value
0230  AND #1 ;is bit 0 = 1?
0240  BEQ UP ;no - 11,12 or 1 o'clock
0250  LDA STICK ;get it again
0260  AND #2 ;is bit 1 = 1?
0270  BEQ DOWN ;no - 5,6 or 7 o'clock
```

```
0280 SIDE LDA STICK ;get it again
0290 AND #4 ;is bit 3 = 1?
0300 BEQ LEFT ;no - 8,9 or 10 o'clock
0310 LDA STICK ;get it again
0320 AND #8 ;is bit 4 = 1?
0330 BEQ RIGHT ;no - 2,3 or 4 o'clock
0340 RTS ;joystick straight up
```

As you can be see, after the mandatory PLA to keep BASIC happy, reading the joystick is just a matter of loading the accumulator from the hardware location STICK ($D000) and then ANDing it with 1, 2, 4, or 8. The ATARI joysticks set one or more of the lower four bits in location $D000 to zero if the stick is pressed in that direction: bit zero for up, bit 1 for down, bit 2 for left, and bit 3 for right. If none of the 4 bits is set to zero, the joystick is in the straight-up position. Note that the joystick may not be simultaneously pressed right and left or up and down, but it may be right and down simultaneously, or left and up.

This program won't work, as you've probably already noticed, since there are 4 undefined labels, UP, DOWN, LEFT, and RIGHT. We will add these routines shortly to produce a machine language subroutine which will not only read the joystick, but also move a player around the screen in response to the joystick direction.

First, note that each of the references to a label for the direction of the joystick uses the BEQ instruction. This says, in effect, that if the result of ANDing a bit forced to 1 with the value found in STICK is a zero, the joystick is pressed in that direction. Think about that. We know that for the result to be zero, in one or both of the numbers each bit must be equal to zero. In the numbers 1, 2, 4, and 8, every bit but one is equal to zero; so in the number stored in STICK, that particular bit must be equal to zero if the result of the AND operation equals zero. For a pictorial example, let's look at the AND operation with 4, with the joystick pressed in different directions:

Joystick	STICK	AND with	76543210
right	248	-	11110111
	-	4	00000100
		Result =	00000100

which is not equal to zero. Since the stick was pressed right and ANDing with 4 tests for pressing the joystick left, this is a correct result. Now, another example:

Joystick	STICK	AND with	76543210
left	244	-	11111011
	-	4	00000100
		Result =	00000000

which is equal to zero, showing that the test works correctly. It should be emphasized that any of the three left positions of the joystick would have worked, because they all have a zero as bit 2, so all will AND with 4 to produce a result of zero. In fact, the three joystick positions to the left have the following bit patterns:

8 o'clock	11111001
9 o'clock	11111011
10 o'clock	11111010

It's worth mentioning here that the upper 4 bits of this location reflect the position of a joystick plugged into the second port on your ATARI, in exactly the same way as the lower 4 bits reflect the position of joystick 0.

There are, of course, several other ways of writing the above code. Perhaps one which has occurred to you is to use a subroutine for each direction, as in this excerpt:

```
0210  PLA
0220  LDA STICK
0230  AND #1
0240  BNE D1
0250  JSR UP
0260 D1 LDA STICK
0270  AND #2
0280  BNE D2
0290  JSR DOWN
```

This type of construction would work fine but for one problem: the code is fixed. The locations UP, DOWN, LEFT, and RIGHT have to be within the program, and if we use JSRs to access these routines, we will end up with nonrelocatable code. If that creates no problem for you, then write the routine using JSRs. However, since one of our goals in this book is to make as many of the routines as we can relocatable, we will use the demonstrated construction.

How do we move players around the screen using player-missile graphics? Horizontal movement is easy. All we have to do is POKE the desired horizontal position into the horizontal position register for that particular player, who will appear there instantly. In the case of the first player, player zero, the horizontal position register is located at $D000. We'll call it HPOSP0, and we'll need to add line 170 to the above code:

```
0170 HPOSP0 = $D000
```

which will enable us to refer to this location using the label name.

What about vertical motion? To move a player vertically, we actually have to move each byte of the player to a new location, which is why we needed to set up the indirect address for the Y position. We'll use a technique we've already we used to move the character set. But in this case we can get by with only one indirect address, since we're moving the player only 1 byte away from its current address. We'll assume that the player is 8 bytes high and appears as a hollow square. If we want to move the player up the screen 1 byte, we'll need to begin by moving the top byte first, and so on down the player. If we try to move the lower byte first, it will overwrite the next higher byte and we'll lose that higher byte. Pictorally, it will look like:

```
before move     after move
...........     ...........
...........     .xxxxxxxx..
xxxxxxxx...     .x......x..
x......x...     .x......x..
x......x...     .x......x..
x......x...     .x......x..
x......x...     .x......x..
x......x...     .x......x..
x......x...     .xxxxxxxx..
xxxxxxxx...     ...........
...........     ...........
```

Conversely, to move the player down the screen 1 byte, we'll need to begin by moving the bottom byte first, and we'll work our way up the player.

There's one final problem. In the picture above, the bottom of the player would really be 2 bytes high after being moved, since although we have placed a copy of the bottom byte in the correct position 1 byte higher, we have not moved anything into the space originally occupied by this bottom byte. If we don't correct for this problem, the new figure will look like this:

```
........
........
xxxxxxxx
x......x
x......x
x......x
x......x
x......x
x......x
xxxxxxxx
xxxxxxxx
........
```

In fact, if we don't correct for this, as we move the figure up the screen, we'll leave a tail dangling behind the figure, a clever effect, but not what we intended at all!

Fortunately, there is an easy way to solve this problem; just move 1 byte more than is in the player. Note that since the player is 8 bytes high, if we move 9 bytes, we'll be moving a zero byte into the space formerly occupied by the bottom of the player; so the new player will still have a single line at the bottom, instead of the double line pictured above. Obviously, when we are moving the player down the screen, we can also move 9 bytes instead of 8, solving the problem there, as well. Now that we know how to move the players both horizontally and vertically, let's look at the whole routine, and then we'll describe it in detail.

```
0100 ; ******************************
0110 ;initialize locations
0120 ; ******************************
```

```
0000          0130          *=   $600        ;safe place for routine
00CC          0140 YLOC     =    $CC         ;indirect addr. for Y
00CE          0150 XLOC     =    $CE         ;to remember X position
D300          0160 STICK    =    $D300       ;hardware STICK(0) location
D000          0170 HPOSP0   =    $D000       ;horizontal pos. P0
              0180 ; *****************************
              0190 ;now read the joystick #1
              0200 ; *****************************
0600 68       0210          PLA             ;keep the stack neat
0601 AD00D3   0220          LDA  STICK      ;get joystick value
0604 2901     0230          AND  #1         ;is bit 0 = 1?
0606 F016     0240          BEQ  UP         ;no - 11,12 or 1 o'clock
0608 AD00D3   0250          LDA  STICK      ;get it again
060B 2902     0260          AND  #2         ;is bit 1 = 1?
060D F020     0270          BEQ  DOWN       ;no - 5,6 or 7 o'clock
060F AD00D3   0280 SIDE     LDA  STICK      ;get it again
0612 2904     0290          AND  #4         ;is bit 3 = 1?
0614 F02E     0300          BEQ  LEFT       ;no - 8,9 or 10 o'clock
0616 AD00D3   0310          LDA  STICK      ;get it again
0619 2908     0320          AND  #8         ;is bit 4 = 1?
061B F02F     0330          BEQ  RIGHT      ;no - 2,3 or 4 o'clock
061D 60       0340          RTS             ;joystick straight up
              0350 ; *****************************
              0360 ;now move player appropriately
              0370 ;starting with upward movement
              0380 ; *****************************
061E A001     0390 UP       LDY  #1         ;setup for moving byte 1
0620 C6CC     0400          DEC  YLOC       ;now 1 less than YLOC
0622 B1CC     0410 UP1      LDA  (YLOC),Y   ;get 1st byte
0624 88       0420          DEY             ;to move it up one position
0625 91CC     0430          STA  (YLOC),Y   ;move it
0627 C8       0440          INY             ;now original value
0628 C8       0450          INY             ;now set for next byte
0629 C00A     0460          CPY  #10        ;are we done?
062B 90F5     0470          BCC  UP1        ;no
062D B0E0     0480          BCS  SIDE       ;forced branch!!!
              0490 ; *****************************
              0500 ;now move player down
              0510 ; *****************************
062F A007     0520 DOWN     LDY  #7         ;move top byte first
0631 B1CC     0530 DOWN1    LDA  (YLOC),Y   ;get top byte
0633 C8       0540          INY             ;to move it down screen
```

```
0634 91CC    0550        STA   (YLOC),Y  ;move it
0636 88      0560        DEY             ;now back to starting value
0637 88      0570        DEY             ;set for next lower byte
0638 10F7    0580        BPL   DOWN1     ;if Y>=0 keep going
063A C8      0590        INY             ;set to zero
063B A900    0600        LDA   #0        ;to clear top byte
063D 91CC    0610        STA   (YLOC),Y  ;clear it
063F E6CC    0620        INC   YLOC      ;now is 1 higher
0641 18      0630        CLC             ;setup for forced branch
0642 90CB    0640        BCC   SIDE      ;forced branch again
             0650 ; ****************************
             0660 ;now side-to-side  left first
             0670 ; ****************************
0644 C6CE    0680 LEFT   DEC   XLOC      ;to move it left
0646 A5CE    0690        LDA   XLOC      ;get it
0648 8D00D0  0700        STA   HPOSP0    ;move it
064B 60      0710        RTS             ;back to BASIC - we're done
             0720 ; ****************************
             0730 ;now right movement
             0740 ; ****************************
064C E6CE    0750 RIGHT  INC   XLOC      ;to move it right
064E A5CE    0760        LDA   XLOC      ;get it
0650 8D00D0  0770        STA   HPOSP0    ;move it
0653 60      0780        RTS             ;back to BASIC - we're done
```

Let's look at the construction of the program as a whole — the program flow. We first test to see if the joystick is pressed up. If it is up, we branch to UP. If not, we test for down, and if it's down, we branch to DOWN. In either of these cases, after moving the player, we need to go back to test for side-to-side movement, since it is possible to move both horizontally and vertically simultaneously. This branch back to test for horizontal movement is accomplished by forced branches in lines 480 and 640. In line 480, the carry bit must be set, since if it were not, line 470 would have branched away from line 480. In line 640, the forced branch is even more obvious, since in line 470, we clear the carry bit and then branch if the carry bit is clear, as we know it must be! Why not just jump back to SIDE? Again, because we want the routine to be relocatable, and if we use any JMP commands, it will not be. This technique of the forced branch is common in relocatable code, and is fairly easy to accomplish, now that you know how.

Once we've tested for both horizontal and vertical movement, we're done and can return to BASIC. Note that this routine contains three RTS instructions. There's no rule about a routine having only one RTS; whatever works, do! In this case, we can return if the stick is vertical (line 340) or if we've moved the player left (line 710) or right (line 780), since in any of these three cases, we've exhausted the possibilities, testing for every combination of movements.

The specific code for moving right or left reads the current X coordinate from its storage location, incrementing or decrementing it as appropriate, and stores it in both its storage location again and the horizontal position register for player zero, HPOSP0.

Now we'll discuss vertical motion. Since moving the player up the screen results in a Y position 1 unit less than its initial value (the lower the Y coordinate, the higher the player appears on the screen), we will need to eventually decrement the YLOC value. We can take advantage of this decrementing if we do it near the beginning of the routine. When YLOC is decremented, it points to the destination of the top byte of the player. Setting Y to 1 allows the command labeled UP1 to point initially to the top byte of the player, in its original location. We then decrement Y, and the next STA instruction puts that byte in its new, higher location on the screen. We must then increment Y twice, once for the decrement we went through and once to get the next byte. We're going to move 9 bytes, and we started with Y = 1, so when Y = 10, we're done. If we are not done, we'll go back up to get the next byte, and if we are done, we'll take the forced branch back up to check for horizontal motion. The technique here is to use indirect addressing for both the LDA and the STA, but changing the offset (Y register) by 1 between the LDA and the STA. That allows us to load from one location and store into another, without a lot of fuss.

We'll use a slighly different algorithm to move a player down the screen. As mentioned above, we begin with the bottom byte, so we set Y equal to 7 (the bytes are 0 to 7 in this case). We LDA indirect, then increment the Y register, and then STA indirect, like we did above, but in this case, we store into a higher location than we load from. We then decrement twice, once for the increment and once to get the next byte, and if Y is still greater than or equal to zero, we keep going. If not, we'll store a zero into the original lowest byte, by incrementing Y to set it back to zero (it had reached −1, or $FF in hexidecimal) and storing a zero into YLOC, indirect. Then we increment YLOC, since we've moved the player down 1 position on the screen, and force a branch back to check for horizontal movement. Note that when we moved up the screen, we actually moved 9 bytes, but when we moved down the screen, we moved 8 bytes, and then stuffed a zero to eliminate the tail of the player. We used two methods in order to show that either works.

By the way, one concern you may have about this routine is that it reads the joystick four separate times. "What happens," you may ask, "if the position of the joystick changes between reads?" If we calculate the time over which all four reads of the joystick occur, we can see that all reading takes place in less than 25 microseconds. Little chance of a change in that time span!

Now that we have our machine language routine, all we need to do is incorporate it into a BASIC program which can use it appropriately. Such a program is given below:

```
10 TOP=PEEK(106)-8:REM Save 8 pages
20 POKE 106,TOP:REM Make room for PMG
30 GRAPHICS 0:REM Reset display list
40 PMBASE=TOP*256:REM Set up PM area
50 POKE 54279,TOP:REM Tell ATARI where PMBASE is
60 INITX=120:REM Initial X position
70 INITY=50:REM Initial Y position
80 POKE 559,46:REM Double line resolution
90 POKE 53277,3:REM Enable PM
100 GOSUB 20000:REM Set up our routine
110 FOR I=PMBASE+512 TO PMBASE+640:REM PM Memory
120 POKE I,0:REM Clear it out
130 NEXT I:REM Could use ERASE$ here!
140 RESTORE 25000:REM Player data is stored here
150 Q=PMBASE+512+INITY:REM Where player will be in memory
160 FOR I=Q TO Q+7:REM Player is 8 bytes high
170 READ A:REM Get player data
180 POKE I,A:REM Put it in proper place
190 NEXT I:REM And so on
200 POKE 53248,INITX:REM Setup X position
210 YHI=INT(Q/256):REM High byte of initial Y position
```

```
220 YLO=(PMBASE+512+INITY)-YHI*256:REM Low byte
230 POKE 204,YLO:REM Tell ML routine where Y is
240 POKE 205,YHI:REM Tell ML routine where Y is
250 POKE 206,INITX:REM Tell ML routine where X is
260 POKE 704,68:REM Make player red
270 Q=USR(ADR(JOYSTICK$)):REM Let's try it!
280 GOTO 270:REM Just loop
20000 DIM JOYSTICK$(87):REM Where to put routine
20010 FOR I=1 TO 87:REM Length of routine
20020 READ A:REM Get a byte
20030 JOYSTICK$(I,I)=CHR$(A):REM Put it into string
20040 NEXT I:RETURN :REM All done
20050 DATA 104,173,0,211,41,1,240,22,173,0
20060 DATA 211,41,2,240,32,173,0,211,41,4
20070 DATA 240,46,173,0,211,41,8,240,47,96
20080 DATA 160,1,198,204,177,204,136,145,204,200
20090 DATA 200,192,10,144,245,176,224,160,7,177
20100 DATA 204,200,145,204,136,136,16,247,200,169
20110 DATA 0,145,204,230,204,24,144,203,198,206
20120 DATA 165,206,141,0,208,96,230,206,165,206
20130 DATA 141,0,208,96,0,208,96
25000 DATA 255,129,129,129,129,129,129,255
```

Line 100, which sets up the subroutine we just wrote, prepares us to call the subroutine in line 270. Note that line 280 just loops back to this subroutine call, so all that this program will do is move the red, hollow square player around the screen. The program could be expanded considerably by adding code from line 280 on, as long as line 270 remains in the main loop of the game. Each time line 270 is accessed, the joystick is read and the player is moved appropriately. Try it! Notice how smooth and even the motion of the player is. Then try a similar program all in BASIC, and watch how the vertical movement turns the player into an inch-worm, slowly crawling up or down the screen.

The bulk of this program sets up player-missile graphics in BASIC. Since virtually all parameters, from the color of the player to its shape and size, are controlled from this BASIC program and not from the machine language subroutine, this routine should merge nicely with almost any program requiring joystick movement of player zero. With simple modifications that you can now try, it will handle other players, other joysticks, or even multiple players and joysticks. You can even try adding missiles, perhaps when the joystick button (monitored by location $D010) is pressed! The only way to really learn assembly language programming is through programming, and what better time to start than now?

PART THREE
APPLICATIONS

CHAPTER EIGHT
THE DISPLAY LIST AND USING INTERRUPTS

THE ANTIC CHIP

In one very important regard, your ATARI computer is unique when compared with most other available microcomputers. Most microcomputers contain a single microprocessor, the 6502 or Z-80 or one of the many others available. Your ATARI, however, has four microprocessors, three of which have been specifically designed by ATARI and are unique to their computers. In this chapter, we will discuss one of these, called ANTIC.

The ANTIC chip in your ATARI computer is responsible for the video display which is such an important feature of ATARI computers. In most other microcomputers, the microprocessor is responsible not only for calculations and program flow, but also for maintaining the video display. ATARI designed the ANTIC chip to relieve the 6502 of this burden, allowing ANTIC to handle the video display and the 6502 to handle the program which is running.

DISPLAY MEMORY

A specific area of RAM is set aside in your ATARI to house the information which your program is to display on the TV or monitor screen. We will call this area of RAM the **display memory**. As with most parts of RAM used for specific purposes in the ATARI computers, display memory has a specific pointer, which can always tell us where display memory is, even if we move it around. Since we may have as much as 48K RAM in a normal ATARI, we need 2 bytes to hold the address of display memory; they are found in locations 88 and 89. In general, whenever you use a GRAPHICS X command, the operating system sets up display memory for graphics mode X just below the top of memory. Since the amount of RAM required for display memory can vary greatly, depending on which graphics mode we have chosen, it is very important to be able to know where in RAM the display memory starts; and these two memory locations can tell us. To determine the beginning of display memory, one line of BASIC is all that's required:

```
10 BEGDM=PEEK(88)+256*PEEK(89)
```

This line converts the high and low bytes of the pointer to the beginning of display memory into a single address. Let's see how we can use this information.

We know that if we issue a GRAPHICS 0 command in BASIC, the screen will clear. What actually happens is that the operating system looks in location 106, which we've used before, to determine where the top of RAM memory is. It then determines how large display memory must be for that particular graphics mode and automatically clears that space in memory, so that when the graphics mode is established, the screen will be clear, and not filled with random garbage. Finally, the display list, which we will discuss shortly, is set up, and control is then passed back to BASIC.

Once we have a GRAPHICS 0 screen set up, we know that we can get the letter A to appear in the upper left-hand corner of the screen by typing

```
PRINT "A"
```

There is another way to accomplish this same end, however. Now that we know where display memory is located in RAM, we can simply POKE the correct value for the letter A into the appropriate part of display memory, and the letter will appear on the screen, just like it does when we PRINT it to the screen.

```
POKE BEGDM+2,33
```

The +2 in this command allows for the left margin of 2 which is the default left margin on ATARI computers. The 33 stands for the character A in display code. Note that your ATARI actually keeps three separate sets of codes for the meaning of the 256 possible values of the ASCII codes. The first is ATASCII, or ATARI ASCII code, which is used in BASIC; for example:

```
PRINT CHR$(65)
```

which will print the letter A to the screen. The second set of codes is the **display set**, in which the letter A corresponds to a code of 33, as we saw above. This is the code set used when storing information directly into display memory. The third set is called the **internal character set**; it is used when your ATARI reads the keys of your keyboard, for instance. The most common use of the internal character set is when you would like to know what key was last pressed. Location 764 is a 1-byte buffer which contains the internal code of the last key pressed. If location 764 contains a 255, no key has been pressed. To wait for a key to be pressed, we can write this:

```
100 POKE 764,255
110 IF PEEK(764)=255 THEN 110
```

If we want to know which key was pressed, we have to refer to the internal character set. For instance, if PEEK(764) = 127, then the capital letter A was the last key pressed.

The three character sets used in your ATARI are listed for reference in Appendix 2. We could do all PRINTing to the screen by referring to this list and POKEing the appropriate display codes into the proper place in display memory, as we did with the letter A above.

Let's try an experiment. We'll POKE the same code, 33, into display memory, but instead of using a GRAPHICS 0 screen, we'll try other graphics modes.

```
10 FOR MODE=0 TO 8:REM The graphic modes
20 GRAPHICS MODE:REM Set the mode=MODE
30 BEGDM=PEEK(88)+256*PEEK(89):REM Where is display memory?
40 POKE BEGDM+2,33:REM POKE display character A there
50 FOR DELAY=1 TO 700:NEXT DELAY:REM Give a chance to see display
60 NEXT MODE:REM Now for the next mode
```

When we run this program, we see something very interesting happen. First of all, in GRAPHICS 0, the expected letter A appears in the upper left-hand corner of the screen. In GRAPHICS 1 and 2, moderate- and large-sized yellow letter A's appear in that position, respectively. However, in the other graphics modes, no letter A appears at all, and we just see dots of various colors!

THE DISPLAY LIST

The reason for these differences between the graphics modes lies in the way display memory is interpreted by ANTIC. If ANTIC just took whatever was in display memory and put it on the screen, it wouldn't be saving the 6502 very much work at all. The 6502 would still have to figure out what the display should look like and then arrange display memory appropriately, all of which would take a great deal of time. Therefore, in the ATARI, the ANTIC chip does this work for the 6502. All the 6502 has to do is set up a short program which the ANTIC chip can understand, telling ANTIC how the 6502 wants the display memory interpreted, and ANTIC does the rest. This program is called the **display list**. To fully understand the capabilities this display list gives us as programmers, we'll need to learn a new programming language. Fortunately, there aren't many instructions in this language, so it's pretty easy to learn.

We'll list the instructions here, in both decimal and hexadecimal notation for versatility, and then describe each instruction in detail.

Hex.	Decimal	Instruction
0	0	Leave 1 blank display line
10	16	Leave 2 blank display lines
20	32	Leave 3 blank display lines
30	48	Leave 4 blank display lines
40	64	Leave 5 blank display lines
50	80	Leave 6 blank display lines
60	96	Leave 7 blank display lines
70	112	Leave 8 blank display lines
2	2	Display as GRAPHICS 0 text mode
3	3	Display as special text mode
4	4	Display as 4-color text mode
5	5	Display as large 4-color text mode
6	6	Display as GRAPHICS 1 text mode
7	7	Display as GRAPHICS 2 text mode
8	8	Display as GRAPHICS 3 4-color graphic mode
9	9	Display as GRAPHICS 4 2-color graphic mode
A	10	Display as GRAPHICS 5 4-color graphic mode
B	11	Display as GRAPHICS 6 2-color graphic mode
C	12	Display as special 160x20, 2-color graphic mode
D	13	Display as GRAPHICS 7 4-color graphic mode
E	14	Display as special 160x40, 4-color graphic mode
F	15	Display as GRAPHICS 8, 1 1/2 color graphic mode
1	1	Jump to location specified by next two bytes
41	65	Jump to location specified by next two bytes and wait for vertical blank

Four more instructions can be included by setting 1 of 4 bits in the instruction code to a 1. These are:

Bit	Instruction
4	Enable fine vertical scrolling
5	Enable fine horizontal scrolling

6 Load memory scan from next two bytes
7 Set a display list interrupt for the next line

Whew! Seems like a lot, all at once, but if we take it one step at a time, it will be fairly easy. We'll begin by looking at a simple display list. This can be done fairly easily, since, like display memory, the display list has a pointer, found in memory locations 560 and 561, which can always tell us where the display list is located. That makes it easy to write a simple BASIC program to print the display list to the screen so we can have a look at it.

```
10 GRAPHICS 0:REM Simple display list
20 DL=PEEK(560)+256*PEEK(561):REM Address of display list
30 FOR I=DL TO DL+31:REM Length of display list
40 PRINT PEEK(I);" ";:REM Skip one space between bytes
50 NEXT I:REM Finished printing it
```

If we run this program, our screen should show something like the following:

```
112 112 112 66 64 156 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
65 32 156
```

If you have less than 48K of memory in your computer, the last 2 bytes, and the fifth and sixth bytes may differ from those, as we'll see. Let's dissect this display list one byte at a time, remembering that this display list is a computer program and that the computer in this case is ANTIC. Looking at our list of instructions above, we see that 112 means to leave 8 blank display lines. Since there are 3 112's, that would seem to mean that the beginning of this program is telling ANTIC to leave 24 blank display lines on the screen. Can this be right?

Most televisions are designed to overscan the visible screen. You may have noticed that on some sets, the output from your computer seems to start closer to the top or closer to the left or right side of the screen than on others. To allow for this difference between TV sets, most display lists begin with these 24 blank display lines. Of course, we need to remember that in GRAPHICS 0, each character is 8 bytes high. Therefore, 24 blank display lines is exactly the amount of space that three lines of GRAPHIC 0 text would occupy. Similarly, the normal screen in GRAPHICS 0 contains 192 display lines (8 times 24). You are free to add another line or two of text to customize a display list, but although it may work fine on your own TV or monitor, it may not work as well on someone else's set.

The next 3 bytes of the display list were 66, 64, and 156. When we look at the set of possible instructions for ANTIC given above, we don't see 66 listed at all. This 66 is a sum of 64 and 2. The 64 is derived from setting bit 6 of the ANTIC instruction 2. This byte tells ANTIC that we want a line of GRAPHICS 0 displayed here, and, since bit 6 is set, that we also want to load memory scan at this point. **Load memory scan** means that the next 2 bytes of the display list are a pointer to where ANTIC can find the display memory for that line, and all succeeding lines, until a new load memory scan instruction is encountered. The 2 bytes 64 and 156 are in typical LSB, MSB 6502 order, and to translate them to an address, we add the LSB to 256 times the MSB. Since 64 + 256 * 156 = 40000, we know, as does ANTIC now, that display memory can be found at 40000 and above. The next 23 bytes of the display list are all 2's, and simply tell ANTIC that we want all GRAPHICS 0 lines, consisting of 40 bytes of text per line, each byte 8 display lines high. With the line specified by the load memory scan instruction, that totals 24 lines of GRAPHICS 0, or a normal GRAPHICS 0 screen. The next instruction of the display list is 65, which translates to jump and wait for the vertical blank.

THE VERTICAL BLANK

To understand the vertical blank instruction, we must first discuss the method by which the picture on your TV screen is produced. The inner front surface of the picture tube is coated with phosphors, chemicals which emit light when struck by an electron beam. At the rear of the picture tube is an electron gun, which shoots electrons toward the front surface to strike the phosphor-coated surface. The horizontal and vertical position at which this beam of electrons hits the phosphors is controlled by deflecting the beam in a precise way. From the point of view of the person watching the TV, the beam begins in the upper left-hand corner of the screen and traverses a single line across the screen until it reaches the upper right-hand corner. It then jumps back to begin line 2, and so on. The intensity of the beam varies as it scans, producing darker or lighter spots and creating a picture. Devices of this type are called **raster-scan devices**, and are by far the most common system for producing electronic pictures. The other major type of device is the **vector device**, in which the beam of electrons draws a line by beginning at the point of origin of the line, and scanning in any direction the line takes until its end is reached. Raster-scan devices draw a line by drawing the whole screen, on which the line happens to be displayed; vector devices draw only the line.

When a raster-scan device has scanned the entire screen with the electron beam, the position of the beam is returned to the upper left-hand corner, and the device then waits for a synchronization signal, telling it to begin the next screen, or frame. In fact, we can see this pause by adjusting the vertical hold control on a TV set until the picture begins to roll. The wide black horizontal bar which appears to move vertically across the screen is created by the electron beam waiting for the vertical synchronization signal. This interval, during which the electron beam is not scanning across the screen, is called the **vertical blank**.

Your ATARI computer produces 262 scan lines for each picture produced on the screen, and the screen is completely redrawn 60 times every second. This seems very fast, but in relation to the speed of the computer, the drawing of the screen is actually proceeding at a snail's pace. The entire drawing of one screen of a display takes 16,684 microseconds, and the vertical blank interval is about 1400 microseconds. If we remember that 1 machine cycle of the computer is less than 1 microsecond, the relative speeds of the computer and the TV become obvious.

The instruction to jump and wait for the vertical blank, which we encountered above, is actually a 3-byte instruction. It tells ANTIC that its next instruction can be found at the place in memory pointed to by the next 2 bytes, in this case, 32 and 156. This is, in fact, the address of the display list — the same address, found in memory locations 560 and 561, which we discussed above. The instruction to jump and wait for the vertical blank furthermore tells ANTIC not to begin executing the program found at that address until the vertical blank interval is over. This wait accomplishes two things. First, it synchronizes the computer and the TV, so the picture is stable. Second, it gives the computer about 1400 microseconds 60 times per second to use while nothing else is happening. The ATARI uses this time for internal housekeeping, such as updating all of the internal timers and a lot more. We'll discuss some uses for this time later in this chapter.

PICTURE RESOLUTION

One final note about the TV picture produced: although 262 scan lines are produced per frame, only 192 of them are visible on most sets, because of overscan, so the highest vertical resolution of the ATARI is 192 **pixels** (picture elements) in the vertical dimension. In the horizontal dimension, the highest usable resolution is 160 pixels, although GRAPHICS 8 screens actually use 320 pixels of resolution in the horizontal direction. However, in GRAPHICS 8, we are all familiar with the color artifacting which results. If we draw a diagonal line on the screen in GRAPHICS 8, the line appears to be of different colors, depending on its location on the screen. To produce a true color on the screen, two adjacent horizontal pixels should be turned on, or else only one of the primary colors used for broadcast TV may appear when we intended a color such as white to appear. When color rendition is important, our horizontal resolution is limited to 160 pixels.

DIRECT MEMORY ACCESS

Now we are beginning to understand how the picture is produced by an ATARI computer. In summary, a portion of memory is used to store the information which is to be displayed (display memory), and this is interpreted by ANTIC using the program called the display list. One further note about this process: ANTIC and the 6502 actually share the area of RAM called the display memory. The 6502 produces and changes the information stored there, and ANTIC reads it, interprets it, and puts it on the screen. It should be apparent that both microprocessors cannot simultaneously access the same memory. In fact, when the 6502 needs it, ANTIC can't access it, and when ANTIC is reading display memory, the 6502 is turned off. ANTIC accesses display memory by a process called **D**irect **M**emory **A**ccess, or DMA. In doing so, ANTIC actually steals time from the 6502, and during this time no processing is done in the 6502. When ANTIC is finished reading display memory, the 6502 begins processing again. This process of DMA actually slows program execution somewhat; a BASIC program may be speeded up by 30 percent or so by disabling DMA. To disable DMA from BASIC, all that is needed is to

```
POKE 559,0
```

To reenable DMA,

```
POKE 559,34
```

One serious drawback offsets the increase in speed obtained: your TV screen will turn blank and remain off until DMA is reenabled. However, anything PRINTed to the screen during the time DMA is disabled will appear when DMA is reenabled.

Now that we know how the TV picture is produced, we can begin to modify it for our own purposes. Many articles have appeared describing how to create custom display lists, such as those combining several different text modes and perhaps even several lines of graphics as well. The remainder of this chapter will be devoted to programs which cannot be written in BASIC, but which can be accessed from BASIC using machine language subroutines; they will perform some rather interesting tasks for us.

INTERRUPT PROCESSING

Used in the context of this book, an **interrupt** is a message telling the 6502 to stop whatever was about to happen in your

ATARI and instead do something else defined by the programmer. When that task is finished, the 6502 may then continue with whatever it had planned prior to the interrupt. Two types of interrupts are normally used, both of which relate to the TV picture — display list interrupts and vertical blank interrupts. Neither of these can be used without machine language subroutines, since languages such as BASIC are far too slow for these purposes.

DISPLAY LIST INTERRUPTS

First we'll cover display list interrupts. When we discussed the display list, we noted that if bit 7, the most significant bit, of any display list instruction is set (equal to 1), a display list interrupt is enabled for the next scan line of the TV. What does this mean?

On page 2 of RAM, in locations $200 and $201 (decimal 512 and 513), is the display list interrupt vector. A vector, as we have discussed before, is like a signpost, pointing somewhere. Normally, location $200 contains $B3 and location $201 contains $E7, so this signpost points to $E7B3. This location contains the byte $40, which is the machine language code for RTI, Return from Interrupt. Another way of saying this is that the display list interrupt vector normally points to an end to an interrupt routine. This is to prevent you from setting a display list interrupt and having the computer go off to some random address and try to execute the code found there.

Timing considerations are important in the use of display list interrupts. A normal display list interrupt consists of three parts. Part 1 occurs during the time it takes the beam of electrons to finish scanning the line which has bit 7 set. Part 2 occurs between the time that the beam begins scanning the line on which the interrupt takes effect and the time that the beam enters the visible portion of the line. Part 3 begins when the beam enters the visible screen and concludes at the end of the display list interrupt routine.

The electron beam takes 114 machine cycles to scan each horizontal line. Although bit 7 is set at the beginning of the line, the 6502 is not informed about the interrupt until cycle 36. It is therefore apparent that long machine language routines cannot be implemented using display list interrupts; there is just not enough time for them.

A SIMPLE EXAMPLE

Display list interrupts are commonly used to change the background color of the screen in midscreen. Let's write such a routine, and then implement it. Since we will be interrupting the 6502 while it's executing instructions, one thing we must be sure of is that if we plan to use either register or the accumulator, we need to save their initial values and restore those values before returning from the interrupt. Let's look at the program and then discuss it:

```
              0100 ; *****************************
              0110 ;setup of simple DLI routine
              0120 ; *****************************
0000          0130        *=    $600      ;Safe place for routine
D40A          0140 WSYNC  =     $D40A
D018          0150 COLPF2 =     $D018     ;Background color
              0160 ; *****************************
              0170 ;now for the DLI routine
              0180 ; *****************************
0600 48       0190        PHA             ;Save value in accumulator
0601 A942     0200        LDA   #$42      ;For a dark red color
0603 8D0AD4   0210        STA   WSYNC     ;See discussion
0606 8D18D0   0220        STA   COLPF2    ;Put new color in
              0230 ; *****************************
              0240 ;let's restore the accumulator
              0250 ; *****************************
0609 68       0260        PLA             ;Restore it
060A 40       0270        RTI             ;And we're finished
```

The first thing we should notice about this routine is that it doesn't begin with a PLA instruction. In fact, the only PLA instruction in the program is to restore from the stack the original value which was in the accumulator; this value was placed on the stack by line 190 for safekeeping during the execution of this routine. Yet this routine is meant to interact with BASIC, and we know that any USR call from BASIC needs the PLA instruction to remove the number of parameters from the stack.

This apparent error is not going to get us in trouble, since this routine is not meant to be called by a USR call, but rather is accessed directly by the interrupt routine we will set up in our BASIC program shortly. Interrupts need no PLA instruction, since they pass no information to the machine language routine, and therefore the stack remains tidy.

Let's go through this routine in detail. We first load the accumulator with the hexadecimal number $42, which specifies a dark red color in the ATARI color selection system. This number arises from the sum of 16 times the color, added to the luminance. Since the 4 in $42 is 4 sixteens and the 2 is 2 ones, this represents a color of 4 with a luminance of 2. We store this number in the hardware register for the background color used in GRAPHICS 0, found at address $D018, and called COLPF2 in the ATARI equates system. It is important to understand why we use the hardware register and not the normal color register, which is found at decimal address 710.

If we store a number (such as $42) representing a color into the normal color register at location 710, the screen will turn red and remain red until we change the number stored in that location. However, this is not what we intended to do with this routine. We wanted only the bottom portion of the screen to turn red while the top portion remains its normal blue color. We need to know that the hardware register, $D018, is updated from its shadow register, 710, 60 times per second. During each vertical blank interval, your ATARI reads the value stored in location 710 and places this value in the hardware register, $D018. Therefore, 60 times per second, the screen is told to turn blue, since the number stored in 710, which is 148, tells the computer a blue color is desired. Now look at what our routine is doing.

Sixty times per second, between drawing frames of your TV picture, your ATARI is told that the screen background color should be blue. Our routine tells the same hardware register that after a number of lines of the next frame are displayed, the background color should now be red, so it draws the remaining scan lines of that frame with a red background. Then look what happens when that frame is completed and the next vertical blank interval begins. Your ATARI takes 148 and stuffs it into the hardware register, turning the top of the next frame blue again, and our routine turns the bottom of that frame red again, and so on. The net result is that the top of the picture stays blue, and the bottom stays red. If we had used location 710 in line 220 instead of $D018, the whole screen would have remained red.

Between the loading of the accumulator with the color value desired and the storing of this value into the hardware color register, we see line 210, referring to a WSYNC location at $D40A. This is a very important location for display list interrupts.

Picture the electron beam scanning over your TV screen from left to right. Every time it gets to the right edge, it jumps back 1 line lower and begins again at the left edge with the next line. If we are doing something such as changing the color displayed for the background, we want to be sure that the color change occurs at the beginning of a line rather than somewhere in the middle. If we simply stick the new color value into the hardware register, the background color will change wherever the electron beam happens to be when the new value is placed into $D018. To prevent this, line 210 stores a value (any value: the color is simply at hand, so we'll use it) to location WSYNC. It doesn't matter what value is stored here; it's the act of storing any value to this location which triggers the resulting action. Whenever a value is POKEd to WSYNC, the computer simply **W**aits for the horizontal **SYNC**hronization before proceeding. This horizontal synchronization occurs while the electron beam is off the screen, waiting to begin the next line. After synchronization, the computer executes line 220, which stores the desired color into the hardware register. This method ensures that the color change will always take place at the beginning of a scan line and not sometimes in the middle of a line.

The remainder of this program simply restores the original value which was in the accumulator and then returns to whatever was going on before the interrupt, by means of the RTI (return from interrupt) instruction in line 270.

Installation of a display list interrupt routine requires some programming in BASIC, since the display list interrupt routine cannot, by itself, cause the desired color change. Let's look at the BASIC program used to implement this particular routine:

```
10 GOSUB 20000:REM Setup simple DLI routine
20 HIBYTE=INT(ADR(SIMPDLI$)/256):REM Where is our DLI routine?
```

```
30 LOBYTE=ADR(SIMPDLI$)-256*HIBYTE:REM Its low byte
40 POKE 512,LOBYTE:REM Set up low byte of new vector
50 POKE 513,HIBYTE:REM Set up high byte
60 DL=PEEK(560)+256*PEEK(561):REM Where is display list?
70 POKE DL+12,PEEK(DL+12)+128:REM Set display list bit 7
80 POKE 54286,192:REM Enable DLIs
90 END :REM But the color change stays
20000 DIM SIMPDLI$(13):REM Relocatable code in string
20010 FOR I=1 TO 13:REM Length of simple DLI routine
20020 READ A:REM Get a byte
20030 SIMPDLI$(I,I)=CHR$(A):REM Put it into string
20040 NEXT I:RETURN :REM Finished
20050 DATA 72,169,66,141,10,212,141,24,208,104
20060 DATA 64,246,243
```

As you can see, first we set up the routine we just wrote as a string; this is accomplished in the subroutine at lines 20000 to 20060. We next have to calculate where BASIC has stored this string and break down the address into its high and low bytes. We then can tell the computer where the routine is located, so that when it encounters the display list interrupt instruction, it knows where to turn to find the program it must execute at that time. This information can always be found in the ATARI in memory locations 512 and 513, stored in the usual 6502 fashion of low byte first. Therefore, in lines 40 and 50 we place the 2 bytes of our calculated address into memory locations 512 and 513.

Line 60 finds the display list for us, and since we've used these instructions before, we'll not further discuss them here. Line 70 sets the display list interrupt bit, bit 7, on the twelfth byte of the display list. We could just as easily have set the color change further down the screen by saying, for instance, DL + 20 instead of DL + 12. Experiment, and look at the results for yourself. Just remember that the display list interrupt enable bit must be set on a valid instruction of the display list. Don't try to set it on one of the 2 bytes of address pointing to display memory (DL + 4 or DL + 5), or one of the 2 bytes of address pointing to the beginning of the display list (the last 2 bytes of the display list).

Line 80 is critical! Even though we have done everything required to enable display list interrupts, we have not yet told our ATARI that we would like them enabled. We do this in line 80. This instruction is required before display list interrupts will work, and if you have trouble getting display list interrupts to function, check for this line before pulling your machine language code apart looking for a mistake.

A MORE COMPLICATED EXAMPLE: A TABLE-DRIVEN DLI ROUTINE

There are many uses to which display list interrupts can be put. Some of these are:

1. Change color of background.
2. Change color of the characters.
3. Change the character set entirely (by POKEing the address, in pages, into the appropriate hardware register — $D409, not into 756!).
4. Invert the character set — may be useful in drawing playing cards to the screen: draw half, then invert the character set and draw the bottom half (hardware register = $D401).
5. Simulate motion of a horizon via moving DLIs.

Many other uses are possible, limited only by your imagination. We'll give one more example here, simply to show how to implement a more complicated display list interrupt routine. Just remember that time is short, so keep your code as concise and quick as possible.

This example will introduce **table lookup** techniques. We will put a display list interrupt on every line of a GRAPHICS 0 display and change the color of the background behind every line produced. To do this, we will construct a table of colors, each of which will be used for a single line of the display. Therefore, we need to read each value in turn from the table and store it to the background hardware color register at the appropriate time. The next time through, we need to get the next value from the table for the next line of the display. We'll construct our table on page 4, but it could also have been placed on page 6 or elsewhere in protected memory, as we have already discussed. The assembly language display list interrupt routine is shown below:

```
               0100 ; ******************************
               0110 ; set up initial conditions
               0120 ; ******************************
0000           0130        *=    $600
D018           0140 COLPF2 =     $D018
D40A           0150 WSYNC  =     $D40A
0400           0160 OFFSET =     $0400
               0170 ; ******************************
               0180 ; save registers!!
               0190 ; ******************************
0600 48        0200        PHA              ;save the accumulator
0601 98        0210        TYA              ;and the Y register
0602 48        0220        PHA              ;easy way to save it
               0230 ; ******************************
               0240 ; the routine itself
               0250 ; ******************************
0603 AC0004 0260           LDY   OFFSET     ;get initial offset
0606 B90204 0270           LDA   OFFSET+2,Y ;get color from table
0609 8D0AD4 0280           STA   WSYNC      ;wait for horiz. synch.
060C 8D18D0 0290           STA   COLPF2     ;change color
060F EE0004 0300           INC   OFFSET     ;for next color
0612 AD0004 0310           LDA   OFFSET     ;are we done?
0615 CD0104 0320           CMP   OFFSET+1   ;stores # of colors
0618 9005      0330        BCC   SKIP       ;no-exit DLI routine
061A A900      0340        LDA   #0         ;yes
061C 8D0004 0350           STA   OFFSET     ;reset offset counter
               0360 ; ******************************
               0370 ; remember to restore registers!
               0380 ; ******************************
061F 68        0390 SKIP   PLA              ;set up to restore Y
0620 A8        0400        TAY              ;restore Y
0621 68        0410        PLA              ;restore accumulator
0622 40        0420        RTI              ;exit from DLI routine
```

Note the differences between this routine and the previous display list interrupt routine. Since this program uses both the accumulator and the Y register, we'll need to save both of these on the stack. This is done by PHAing the accumulator value, then transferring the Y register to the accumulator and PHAing it onto the stack.

The major difference between the two display list interrupt routines lies in lines 260 to 270 and 300 to 350. We first load the Y register from OFFSET in line 260. The number thus loaded is an offset into the color table, which begins at $402 and continues upward in memory from there. If OFFSET equals 5, then we'll pick the sixth color in the table (remember: the first color is number zero). This becomes the number stored in the hardware background color register at that time. Lines 300 to 350 simply increment OFFSET and determine whether all of the colors have been used. If they have, we reset OFFSET to zero and exit. If not, we simply exit. Note that location $401 (OFFSET + 1) stores the number of colors in the table so that we can determine when we are done.

Since we saved both the Y register and the accumulator, we'll need to restore them both. We do that in lines 390 to 420 just by reversing the process that saved them.

Now let's look at the BASIC program that we can use to access our table-driven display list interrupt routine:

```
10 GOSUB 20000:REM Set up DLI routine in a string
20 HI=INT(ADR(TABLEDLI$)/256):LO=ADR(TABLEDLI$)-HI*256:REM Get
addresses of DLI routine
30 GRAPHICS 0:SETCOLOR 1,0,0:REM Start with black background
40 RESTORE 270:REM Be sure we're reading the right data
50 FOR I=0 TO 27:REM Number of data in table
60 READ A:REM Get a byte
70 POKE 1026+I,A:REM Put the color into the page 4 table
80 NEXT I:REM Finish copying table
90 POKE 1024,0:REM Start with zero offset into table
100 POKE 1025,27:REM Put number of colors here
140 DL=PEEK(560)+256*PEEK(561)+6:REM Normal DL instructions start
with the seventh byte of the display list
150 DLBEG=DL-6:REM The beginning of the display list
160 FOR I=0 TO 2:REM The first 3 bytes are skip 8 scan lines
170 POKE DLBEG+I,240:REM Set DLIs even on the skipped scan lines!!!
180 NEXT I:REM Finish these three
```

```
190 POKE DLBEG+I,194:REM Set a DLI even on the "load memory scan"
instruction
200 FOR I=DL TO DL+22:REM Change all of the 2s to 130s
210 POKE I,130:REM Set DLIs
220 NEXT I:REM Finished
230 POKE 512,LO:POKE 513,HI:REM Tell ATARI where our routine is
240 POKE 54286,192:REM Enable the interrupts
250 LIST :REM Gives us something to look at through the colors
260 END :REM All finished
270 DATA 6,22,38,54,70,86,102,118,134,150,166,182,198,214
280 DATA 230,246,246,230,214,198,182,166,150,134,118,102,86,70
290 DATA 54,38,22,6
20000 DIM TABLEDLI$(35):REM Set up string
20010 RESTORE 20060:REM Be sure we're reading correct data
20020 FOR I=1 TO 35:REM Number of bytes in routine
20030 READ A:REM Get a byte
20040 TABLEDLI$(I,I)=CHR$(A):REM Put byte in place in string
20050 NEXT I:RETURN :REM Finish string
20060 DATA 72,152,72,172,0,4,185,2,4,141
20070 DATA 10,212,141,24,208,238,0,4,173,0
20080 DATA 4,205,1,4,144,5,169,0,141,0
20090 DATA 4,104,168,104,64
```

The subroutine at line 20000 sets up our routine in a string. Next we find out where the string is stored, and break that address into its high and low bytes. The GRAPHICS 0 command ensures that the display list is set up the way we want it, and we make the background color black initially. We then POKE the color values we would like to see on the screen into place in the table on page 4, one byte at a time. By altering the data in line 270, a different pattern of colors can be obtained. Experiment with these numbers — you'll find it quite easy to produce spectacular effects in your programs. Location $400 (decimal 1024) is POKEd with a zero, since we'd like our routine to begin with the first color in the table. If we were to POKE another number here, say 10, the entire spectrum of colors would be shifted up the screen; we'd start with the eleventh color and end with the tenth.

Next we find both the beginning of the display list and the beginning of the instructions for GRAPHICS 0 (a 2 as the display list instruction), and we set the high bit on every instruction in the display list, thereby setting a display list interrupt for every line. Note that we can even set display list interrupts for the first three instructions of the display list, which only tell ANTIC to leave 8 blank scan lines. By using this routine, we'll make each group of eight blank scan lines a different color! In line 230, we tell the computer where our display list interrupt routine is, and then in the next line, we enable the display list interrupts. The LIST command in line 250 simply puts some text on the screen and scrolls it through the colors created by the display list interrupt routine, giving quite a nice effect.

One note about display list interrupt routines: the ATARI computers use WSYNC to create the click accompanying the depression of each key of the keyboard. Therefore, programs which use display list interrupts a great deal, like this one, may be disturbed by pressing keys. The simplest solution to this problem is not to ask for keystrokes in your program if you use display list interrupts frequently. You might, for instance, choose from a menu by use of the joystick, or use the START, SELECT or OPTION keys to make choices. Another note to make is that SYSTEM RESET will, of course, eliminate any display list interrupts which have been set up, since this command sets up a new GRAPHICS 0 display list.

VERTICAL BLANK INTERRUPTS

A second common type of interrupt used in your ATARI is the vertical blank interrupt, discussed above. Using this system, it's possible to perform multiprocessing on an ATARI computer. In multiprocessing, two programs are being processed simultaneously. Although the use of the vertical blank interrupt cannot produce true multiprocessing, it is possible to set up two programs, so that one is processed in normal time, and one is processed during the vertical blank interval. It will appear that both are being executed simultaneously.

One excellent example of a program utilizing such multiprocessing is EASTERN FRONT, written by Chris Crawford and available through ATARI. In this game, you take the part of the

German Army during Operation Barbarossa, the German invasion of Russia during World War II, and the computer takes the part of the Russian Army. The computer "thinks" about its moves during the vertical blank interval and handles your moves during real time. The longer you think about your move, the more vertical blank intervals pass, and so the more time the computer has to determine its moves.

A second common use of the vertical blank interval for multiprocessing shows up in a wide variety of programs currently available for the ATARI. Have you noticed the background music which plays while the games are played? This music doesn't slow the game down at all, because it's being played only in the vertical blank interval. Our next program will show how this is done, with a fairly simple example. Although this routine is relocatable, we will simply POKE the routine onto page 6 to access it. By now, you already know how to convert a routine to a string, and you can do so quite easily with this one if you like.

Two parts are required in any vertical blank interrupt routine. One, of course, is the routine itself. The other is a short routine for installing the vertical blank interrupt routine.

Normally, as each vertical blank inteval occurs, your ATARI vectors to a specific routine which is executed at every such interval. The routine actually is composed of two parts. The first is called the **immediate**, and the second is called the **deferred** vertical blank routine. The vector for the immediate routine is found at $0222. This is a 2-byte address to which the computer jumps in order to execute every immediate vertical blank interrupt routine; it normally points to the service routine beginning at $E45F. This routine terminates by vectoring through locations $0224 and $0225, which contain the address of the deferred vertical blank interrupt routine, normally found at $E462. Diagrammatically, this is as follows:

VBI → $0222 → $E45F → $0224 → $E462 → RTI

All we have to do to insert our own routine in place of ATARI's normal routine is to direct the vector to our routine instead of ATARI's. To do this, we must first decide which routine we want to replace. As we'll discuss later, long vertical blank interrupt routines have to replace the normal routines; there is not enough time to execute both during one vertical blank interval. Since the immediate routine pointed to by the vector at $0222 is responsible for a lot of the upkeep of the computer, such as updating the system clocks, copying the shadow registers, reading the joysticks and much more, it's safer to keep it going normally, and replace the deferred vertical blank routine; so for this example, we'll use the deferred routine.

Any time we change a vector, we have a potential problem. With the vertical blank vector, which is used 60 times per second and may be used at any time in relation to the execution of our program, the potential for encountering this problem is magnified. It can best be described by a simple example. We know that the byte stored at $0222 is $5F, and the byte at $0223 is $E4. Let's assume that we'd like to change this vector to point to $0620 instead of $E45F; first we change location $0222 to $20, and then we change $0223 to 6. Simple, wasn't it? But suppose that between the time we change location $0222 to $20 and the time we begin to change location $0223, our computer hits a vertical blank interrupt. It will vector through the address stored in these 2 locations, which is now $E420 because we've changed 1 byte but not the other. Off goes the computer into never-never land, since there is nothing executable at address $E420. To get around this problem, ATARI has provided its own routine to change the vertical blank vectors and prevent this problem from occurring. To see how it works, let's look at the code required:

```
LDY  #$20     ;low byte of routine
LDX  #$06     ;high byte
LDA  #07      ;for deferred vector
JSR  SETVBV   ;set the vector
RTS           ;all done
```

If we wanted to set up our routine for the immediate vertical blank routine, we would load the accumulator with 6 before JSRing to SETVBV ($E45C). That's all there is to it. Remember that your vertical blank interrupt routine must be in place before using this installation routine. If it's not, the computer will crash within a sixtieth of a second after this routine is executed.

There is, of course, a finite length to each part of the vertical blank interval. The deferred routine is about 20,000 machine cycles long at maximum, and the immediate routine can't be longer than about 2000 machine cycles. If your routine is longer than these limits, the computer will crash, since the TV display and the computer will no longer be able to maintain synchronization.

Now that we've decided to use the deferred vector and we know how to install our routine, let's look at the routine to play some music in the vertical blank interval, and then we'll discuss it in depth.

```
               0100 ; ******************************
               0110 ; the equates we'll use
               0120 ; ******************************
0000           0130        *=     $0600
00C0           0140 COUNT1 =      $00C0
0224           0150 VVBLKD =      $0224
00C2           0160 COUNT2 =      $00C2
E45C           0170 SETVBV =      $E45C
0660           0180 MUSIC  =      $0660
E462           0190 RETURN =      $E462
D200           0200 SND    =      $D200
D201           0210 VOL    =      $D201
               0220 ; ******************************
               0230 ;   PLA to keep the stack clean
               0240 ; ******************************
0600 68        0250        PLA
               0260 ; ******************************
               0270 ;   initialize counters to zero
               0280 ; ******************************
0601 A900      0290        LDA  #0
0603 85C0      0300        STA  COUNT1     ;timing counter for notes
0605 85C2      0310        STA  COUNT2     ;which note is playing
               0320 ; ******************************
               0330 ;   now reset deferred vector
               0340 ; ******************************
0607 A020      0350        LDY  #$20       ;low byte of routine
0609 A206      0360        LDX  #$06       ;high byte of routine
060B A907      0370        LDA  #07        ;we want deferred vector
060D 205CE4    0380        JSR  SETVBV     ;set vector
0610 60        0390        RTS             ;initialization complete
```

```
              0400 ; ******************************
              0410 ;   VBI routine itself
              0420 ; ******************************
0611          0430          *=   $0620
0620 E6C0     0440          INC  COUNT1      ;for timing note
0622 A6C0     0450          LDX  COUNT1      ;is note finished?
0624 E00C     0460          CPX  #12         ;if >=12 it is done
0626 9005     0470          BCC  NO          ;not yet finished
0628 A900     0480          LDA  #0          ;yes, so set volume=0
062A 8D01D2   0490          STA  VOL         ;now note turned off
062D E00F     0500 NO       CPX  #15         ;15/60 seconds gone?
062F B003     0510          BCS  PLAY        ;yes, so play next note
0631 4C62E4   0520          JMP  RETURN      ;no, let it ride
0634 A900     0530 PLAY     LDA  #0          ;reset counter
0636 85C0     0540          STA  COUNT1      ;for timing
0638 A6C2     0550          LDX  COUNT2      ;get correct note
063A BD6006   0560          LDA  MUSIC,X     ;from table
063D 8D00D2   0570          STA  SND         ;set its frequency
0640 A9A6     0580          LDA  #$A6        ;distortion=10 ($A)
0642 8D01D2   0590          STA  VOL         ;volume=6
0645 E6C2     0600          INC  COUNT2      ;setup for next note
0647 A6C2     0610          LDX  COUNT2      ;are we done?
0649 E008     0620          CPX  #8          ;if = 8, we are done
064B 9004     0630          BCC  DONE        ;no
064D A900     0640          LDA  #0          ;yes-reset counter to
064F 85C2     0650          STA  COUNT2      ;   start over again
0651 4C62E4   0660 DONE     JMP  RETURN      ;all finished
              0670 ; ******************************
              0680 ;   TABLE OF MUSICAL NOTES
              0690 ; ******************************
0654          0700          *=   $0660
0660 F3       0710          .BYTE 243,243,217,243,204,243,217,243
0661 F3
0662 D9
0663 F3
0664 CC
0665 F3
0666 D9
0667 F3
```

The initialization routine sets two counters to zero, one for the number of the note to be played and the other for determining the length of the note. It then installs the vector to our routine, in place of the normal deferred vector. The routine itself begins at $0620 (line 430). We first increment the duration counter. If this equals 12, we turn off the note; otherwise, the note remains playing. The note can be turned off by storing a zero into the hardware register controlling the volume of that voice, in line 490.

To leave a short pause between notes, we wait until the counter reaches 15 before beginning the next note. To play a new note, we store a zero into the duration counter and get the number of the next note to be played from the note counter. We then use that number as an offset into the table of notes found at $0660 (line 710). Therefore, if COUNT2 equals 2, the third note will be played. The notes are looked up in the table in line 560 and are played by the following three lines. We then increment COUNT2 for the next note and determine if we're done in lines 610 to 630; if we are, we begin the notes all over again by resetting COUNT2 to zero.

We leave this routine by jumping to RETURN, location $E462, which ends our routine with the normal deferred routine. Had we used the immediate vector for our routine, we would have pointed our exit to $E45F, or, for a really long routine, to $E462, which would have eliminated all of the normal ATARI vertical blank interval processing but gained us a lot of time for our own processing in the vertical blank interval.

Setting up a vertical blank routine in BASIC is quite simple, as we shall now see for our music-playing routine:

```
10 GOSUB 19000:REM Poke in initialization routine
20 GOSUB 20000:REM Poke in VBI routine
30 GOSUB 21000:REM POKE in table of notes to be played
40 X=USR(1536):REM Turn on the music!
50 END :REM Will not turn off the music
19000 RESTORE 19050:REM Be sure to get the correct data
19010 FOR I=1536 TO 1552:REM Length of initialization routine
19020 READ A:REM Get a byte
19030 POKE I,A:REM Put it in place
19040 NEXT I:RETURN :REM All done
19050 DATA 104,169,0,133,192,133,194,160,32,162
19060 DATA 6,169,7,32,92,228,96
20000 RESTORE 20050:REM Be sure to read the right data
20010 FOR I=1568 TO 1619:REM Length of VBI routine
```

```
20020 READ A:REM Get a byte
20030 POKE I,A:REM Put it in place
20040 NEXT I:RETURN :REM Finished
20050 DATA 230,192,166,192,224,12,144,5,169,0
20060 DATA 141,1,210,224,15,176,3,76,98,228
20070 DATA 169,0,133,192,166,194,189,96,6,141
20080 DATA 0,210,169,166,141,1,210,230,194,166
20090 DATA 194,224,8,144,4,169,0,133,194,76,98,228
21000 RESTORE 21050:REM Read the right data
21010 FOR I=1632 TO 1639:REM Length of the music table
21020 READ A:REM Get a byte
21030 POKE I,A:REM Put it into the table
21040 NEXT I:RETURN :REM All done
21050 DATA 243,243,217,243,204,243,217,243
```

This program simply accesses the three subroutines and then USRs to initialize the routine and insert the vector appropriately. The first subroutine POKEs the initialization routine onto page 6, the second POKEs the vertical blank interrupt routine itself onto page 6, and the third POKEs the color table into its proper place on page 6. Line 40 activates the routine through the initialization routine. Voila! You have music to help you through a long programming session. The music will continue to play until you hit SYSTEM RESET, or until you reset the deferred vector to its original value.

The music played by this routine is actually quite limited. All notes must be of the same length; for instance, all quarter notes or all half notes. Furthermore, only one voice is used. Far more complicated routines are available for the ATARI, to allow you to put intricate multivoiced music into your programs. But now you can even write such a routine yourself.

One final note concerning the vertical blank interval: one extremely powerful use of this feature is for reading the joysticks and moving players around the screen. By putting this routine into the vertical blank interval, we can remove one of the most time-consuming parts of most BASIC programs, and allow the computer to read the joysticks and update player positions 60 times per second, without slowing down the real-time action at all. You might want to try converting the joystick routine we wrote in Chapter 7 into one utilized in the vertical blank interval, as an exercise for yourself.

FINE SCROLLING

We have yet to cover the final two bits of the display list instructions: the horizontal and vertical fine scroll enable bits (bits 4 and 5, respectively). The fine scrolling facility enables programmers to produce some of the most interesting and exciting effects on the ATARI — programs which scroll a seemingly endless screen past the player. In fact, one of the nicest examples of fine scrolling is found in EASTERN FRONT, already mentioned for its use of multiprocessing. A detailed map of Eastern Europe can be scrolled over many normal-sized screens; action takes place all over the map, making for an exciting and challenging experience.

We will now cover an example of fine horizontal scrolling and discuss fine vertical scrolling to enable you to write your own vertical fine scrolling routines. Horizontal fine scrolling has one difficulty we must first deal with. As you know by now, a normal GRAPHICS 0 display list contains the ANTIC code 2 for each line, telling ANTIC that we want the next 40 bytes of display memory to be interpreted as text and placed on the screen accordingly. However, a problem arises when we scroll the information on the screen to the left. Let's look at an example to see the problem graphically:

```
Screen Column Number
                    111111111122222222223333333333
          0123456789012345678901234567890123456789
          ----------------------------------------
.

.
Line 5    aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Line 6    bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
Line 7    cccccccccccccccccccccccccccccccccccccccc
.
.
```

As long as we only have 40 characters to display, there is no problem. However, how can we scroll the display window over this information? For instance, if we try to scroll the screen to the right (scroll the information to the left), what will the last character on each line be? Line 5 will now end with a b, line 6 with a c, and so on. This is not true horizontal scrolling, but actually mixed horizontal and vertical scrolling.

In order to achieve true horizontal scrolling, we need a special form of the display list. We need to build a custom display list which has room for more than 40 characters per line, so that when we scroll, we get to see information which was previously hidden off-screen. Fortunately, we already know the techniques required to build such a custom display list. We'll need to have a separate Load Memory Scan option on every line, and we'll need to reserve enough memory for each line to be far more than 40 bytes long. Let's design our display list with each line 250 bytes long, so our display memory will be over 6 times wider than a normal GRAPHICS 0 screen. That will give us plenty of room to scroll. The display looks like this:

```
Screen Column Number
               111111111122222222223333333333
     0123456789012345678901234567890123456789
.
.
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc
.
.
```

Diagrammatically, we can now see that the display memory for each line of the display is wider than the screen itself. This now gives us room to move the screen from side to side over the data, without getting the artificial vertical scrolling of the b's into the a line, and so on.

The second feature of our display list which we must consider is that each LMS instruction must also have bit 4 set, so we must add 16 to the LMS instruction 64. Of course, we must also add the ANTIC instruction for the interpretation of the data. In this case, we'll use GRAPHICS 0 (ANTIC mode 2), so we'll need to add 2 to this sum. The total of these is 64 + 16 + 2, or 82, which will be the instruction for every line of our custom display list.

We could, of course, construct our modified display list from BASIC, much as we saw above, but let's experiment and write an assembly language program to construct this display list for us. We'll locate the new display list on page 6, where it will be safe. Remember, each display list begins with 24 blank scan lines, then continues with 24 lines of ANTIC codes before the JVB instruction that terminates the list. Let's look at a program which can construct such a display list for us:

```
             0100 ; ****************************
             0110 ; origin and equates
             0120 ; ****************************
0000         0130        *=    $600       ;must be in string
0600         0140 DLIST  =     $0600      ;where DL will be
0058         0150 SAVMSC =     $58        ;disp.mem.addr.
0230         0160 SDLSTL =     $230       ;DL address
E45C         0170 SETVBV =     $E45C      ;to set VB vector
             0180 ; ****************************
             0190 ; initialization routine to set
             0200 ;  up new display list and
             0210 ;  insert the scrolling routine
             0220 ;  into the vertical blank
             0230 ;  interrupt
             0240 ; ****************************
0600 68      0250 INIT   PLA              ;keep stack neat
0601 A970    0260        LDA   #$70       ;8 blank scan lines
0603 8D0006  0270        STA   DLIST      ;  into the first
0606 8D0106  0280        STA   DLIST+1    ;  3 lines of the
0609 8D0206  0290        STA   DLIST+2    ;  display list
060C A018    0300        LDY   #24        ;# of lines in DL
060E A203    0310        LDX   #3         ;set counter
0610 A952    0320        LDA   #82        ;LMS+GRAPHICS 0+scroll
0612 9D0006  0330        STA   DLIST,X    ;into display list
0615 E8      0340        INX              ;keep counter going
0616 A558    0350        LDA   SAVMSC     ;get disp.mem.addr.
0618 9D0006  0360        STA   DLIST,X    ;into display list
061B E8      0370        INX              ;keep counter going
061C A559    0380        LDA   SAVMSC+1   ;get high byte
061E 38      0390        SEC              ;set up for subtract
061F E918    0400        SBC   #24        ;make room for display
0621 9D0006  0410        STA   DLIST,X    ;into display list
0624 E8      0420        INX              ;keep counter going
```

```
0625 88                  DEY                ;one line finished
0626 A952         LOOP   LDA   #82          ;LMS+horiz.scroll
0628 9D0006              STA   DLIST,X      ;into display list
062B E8                  INX                ;keep counter going
062C BDFD05              LDA   DLIST-3,X    ;get last memory
062F 18                  CLC                ;set up for addition
0630 69FA                ADC   #250         ;line is 250 bytes
0632 9D0006              STA   DLIST,X      ;into display list
0635 E8                  INX                ;keep counter going
0636 BDFD05              LDA   DLIST-3,X    ;get high byte
0639 6900                ADC   #0           ;see discussion
063B 9D0006              STA   DLIST,X      ;into display list
063E E8                  INX                ;keep counter going
063F 88                  DEY                ;another line done
0640 D0E4                BNE   LOOP         ;finished? NO
0642 A941                LDA   #65          ;YES;JVB instruct.
0644 9D0006              STA   DLIST,X      ;into display list
0647 E8                  INX                ;keep counter going
0648 A900                LDA   #0           ;page 6 low byte
064A 9D0006              STA   DLIST,X      ;into display list
064D 8D3002              STA   SDLSTL       ;tell ATARI also
0650 E8                  INX                ;keep counter going
0651 A906                LDA   #6           ;page 6 high byte
0653 9D0006              STA   DLIST,X      ;into display list
0656 8D3102              STA   SDLSTL+1     ;tell ATARI
                  ; *****************************
                  ; insert scrolling routine into
                  ; the deferred vertical blank
                  ; *****************************
0659 68                  PLA                ;get routine's addr
065A AA                  TAX                ;to X register
065B 68                  PLA                ;finish address
065C A8                  TAY                ;to Y register
065D A907                LDA   #$07         ;deferred vector
065F 205CE4              JSR   SETVBV       ;set the vector
0662 60                  RTS                ;all finished
```

We begin by putting the three lines that each mean to leave 8 blank scan lines, $70, at the top of our new display list. Next, we load the Y register with 24; we'll use this to keep track of how many lines of the display list we've constructed. The X register is set to 3, since we want to skip over the first three $70 instructions. The next instruction we need in the display list is 82, so we store it appropriately in lines 320 and 330. We then increment our X counter, since we have added a byte to the growing display list. We'll need to increment this counter with each byte added.

Since each line is an LMS instruction, the next 2 bytes of the line must be the address of display memory from which ANTIC will get the information to display. The beginning of the display list should point to the beginning of display memory, and this pointer is always found at locations $58 and $59, SAVMSC. However, our greatly expanded display memory, over 6 times larger than normal, requires a place to reside. Therefore we'll subtract 24 pages from the high byte of the normal location of screen memory, which will give us the room we need for the display memory. The transfer of the information from location $58 to the new display list is accomplished in lines 350 to 370, and the transfer from $59 and enlargement of display memory is done in lines 380 to 420. Since we've now completed a line of the new display list, which contains an LMS instruction and an address, we decrease our line counter, Y, in line 430.

Now we'll enter a big loop, from line 440 to line 570. The loop will be executed 23 times, and each time it will create one more line of our new display list. The first instruction placed into it is 82, as was mentioned above. Then we retrieve the low byte of the last address and add 250 to it, in lines 470 to 510. Remember to use the CLC instruction before any addition! This sets the low byte of the second line of display memory 250 bytes higher than the former line, so each line will be 250 bytes long instead of the normal 40 bytes.

Lines 520 to 540 don't seem to do anything, do they? They add zero to a number, and replace it in memory. But remember, the carry bit is added into each ADC instruction, and we have not cleared the carry since the last addition. Therefore, if the previous addition resulted in a number larger than 255, the low address placed into the display list in line 500 will actually be the sum minus 256. However, the carry would then have been set, and it will increase the high byte of the address by 1 when we add zero. The address will then point to the correct area of memory. There's another way to code this operation:

```
LDA ADDR1
CLC
ADC #250
STA ADDR1
BCC PASS
INC ADDR2
PASS ....
```

In this case, if the carry isn't set by the first add, ADDR2 isn't incremented; but if the first sum is greater than 255, ADDR2 will be 1 higher than it was.

We conclude the loop by decrementing our line counter, Y, again. If Y has not yet reached zero, we have more work to do, and we loop back up to do it. If Y has reached zero, we're done with this part, and we simply need to set up the JVB instruction to point to our display list on page 6. We do this in lines 580 to 670. Note lines 530 and 670. These insert the address of our new display list into locations $230 and $231, the internal pointers to the display list that the ATARI (and ANTIC) uses.

To make the scrolling fast and smooth, we'll place our routine into the vertical blank interrupt. Our BASIC program will pass the address of the scrolling routine to the set-up routine, and lines 720 to 770 pull this address off the stack and set up the scrolling routine in the deferred vertical blank. Finally, we'll return to BASIC in line 780.

Now that we have our display list constructed, all that we need to do is write a short machine language routine which will handle the scrolling itself. To better understand this routine, let's first discuss the mechanism of fine scrolling. A character in GRAPHICS 0 is 8 bits wide. Coarse scrolling is accomplished one character at a time; with each move, every letter on the screen appears to jump 1 position left or right. We want fine scrolling, in which each move should ideally be only 1 pixel, or 1 bit, in either direction. The ATARI lets us accomplish this fairly easily with a register called HSCROL ($D404). The corresponding vertical scroll register, which works in exactly the same way, is called VSCROL and is located at $D405.

HSCROL can accomplish a bit-by-bit scroll of a character for 8 bits, but then it must be reset. If a zero is written to HSCROL, the position of the character is normal. If we write a 1 to HSCROL, the character shifts 1 pixel left. Writing a 2 shifts the image 1 more pixel, and so on, up to 7. At this point, we write another 0 to HSCROL, and we shift the whole character 1 whole position to the left on the screen by changing the address in the LMS instruction on each line. Pictorially, the characters shift like this:

```
Number written to HSCROL
   0          1         2
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
.......I   ......I.   .....I..
```

After we have completed a full cycle from 0 to 7, shifting the character by 1 full position, we can start a new cycle from 0 to 7, and so on. By continuing this, we can scroll the full width of display memory. In fact, the routine we'll write below won't even check the width of memory, so it will continue to fine-scroll all the way to the top of memory if you let it run long enough. You'll get a look at the operating system of your ATARI in a new and completely unique way!

Now that we know what we'll be doing, let's see the program:

```
          0100 ; ******************************
          0110 ; set up equates and origin
          0120 ; ******************************
0000      0130        *=     $600
0600      0140 DLIST  =      $600
D404      0150 HSCROL =      $D404
E462      0160 XITVBV =      $E462
          0170 ; ******************************
          0180 ; save accumulator and X reg.
          0190 ; ******************************
0600 48   0200        PHA           ;save accumulator
0601 8A   0210        TXA           ;transfer X register
```

```
0602 48     0220        PHA             ;and save it
            0230 ; ******************************
            0240 ; do the fine scrolling first
            0250 ; ******************************
0603 A207   0260        LDX  #7         ;8 bits per character
0605 8E04D4 0270 LOOP   STX  HSCROL     ;scroll the 1st
0608 CA     0280        DEX             ;set up for next scroll
0609 10FA   0290        BPL  LOOP       ;loop until 8 are done
060B A207   0300        LDX  #7         ;reset scroll register
060D 8E04D4 0310        STX  HSCROL     ;   to beginning
            0320 ; ******************************
            0330 ; now we'll coarse scroll one
            0340 ; ******************************
0610 A200   0350        LDX  #0         ;counter
0612 BD0406 0360 LOOP2  LDA  DLIST+4,X ;get disp.mem.
0615 18     0370        CLC             ;before addition
0616 6901   0380        ADC  #1         ;raise it by 1
0618 9D0406 0390        STA  DLIST+4,X ;in display list
061B BD0506 0400        LDA  DLIST+5,X ;get high byte
061E 6900   0410        ADC  #0         ;add carry in
0620 9D0506 0420        STA  DLIST+5,X ;in display list
0623 E8     0430        INX             ;move forward in
0624 E8     0440        INX             ;   display list
0625 E8     0450        INX             ;   3 bytes
0626 E048   0460        CPX  #72        ;24*3=72
0628 90E8   0470        BCC  LOOP2      ;not finished
            0480 ; ******************************
            0490 ; now restore registers
            0500 ; ******************************
062A 68     0510        PLA             ;first, X reg.
062B AA     0520        TAX             ;restored
062C 68     0530        PLA             ;then accumulator
062D 4C62E4 0540        JMP  XITVBV     ;exit from VB
```

Since this will be in the vertical blank interrupt, in lines 200 to 220 we'll save both of the registers we'll be using, the accumulator and the X register. Next, in lines 260 to 310 we'll quickly loop through all 8 bits stored into HSCROL, resetting our counter to 7 before we leave. Then in lines 350 to 470 we enter another loop, which simply goes through the display list and raises each address 1 byte, accomplishing the coarse horizontal scroll. If we were scrolling vertically, we would have to add 250 to each address, in order to coarse-scroll up 1 line here (or add 40, if we were using a normal-width display memory). In this loop, we are using the X register as a byte counter rather than as a line counter, so we must increment X 3 times for each loop (since there are 3 bytes per line of the display list).

Finally, in lines 510 to 530, we restore the registers we saved at the beginning of the program, and in line 540 we exit to the exit routine of the deferred vertical blank.

We can now write a very simple BASIC program to use the two routines we have written:

```
10 GOSUB 20000:REM Sets up string to form modified display list
20 GOSUB 30000:REM Sets up string with scrolling routine in it
30 FOR I=34000 TO 40000 STEP 5:POKE I,86:NEXT I:REM Puts lines
into display memory so we can see the scroll
40 DUMMY=USR(ADR(DLSCROLL$),ADR(SCROLL$))
50 GOTO 50
20000 DIM DLSCROLL$(99):REM Length of routine to set up scrolling
display list
20010 FOR I=1 TO 99:REM Length of string
20020 READ A:REM Get a byte
20030 DLSCROLL$(I,I)=CHR$(A):REM Insert it into string
20040 NEXT I:RETURN :REM All finished
20050 DATA 104,169,112,141,0,6,141,1,6,141
20060 DATA 2,6,160,24,162,3,169,82,157,0
20070 DATA 6,232,165,88,157,0,6,232,165,89
20080 DATA 56,233,24,157,0,6,232,136,169,82
20090 DATA 157,0,6,232,189,253,5,24,105,250
20100 DATA 157,0,6,232,189,253,5,105,0,157
20110 DATA 0,6,232,136,208,228,169,65,157,0
20120 DATA 6,232,169,0,157,0,6,141,48,2
20130 DATA 232,169,6,157,0,6,141,49,2,104
20140 DATA 170,104,168,169,7,32,92,228,96
30000 DIM SCROLL$(48):REM Length of routine
30010 FOR I=1 TO 48:REM Get it all
30020 READ A:REM Get a byte
30030 SCROLL$(I)=CHR$(A):REM Put it into string
30040 NEXT I:RETURN :REM All done
30050 DATA 72,138,72,162,7,142,4,212,202,16
30060 DATA 250,162,7,142,4,212,162,0,189,4
```

```
30070 DATA 6,24,105,1,157,4,6,189,5,6
30080 DATA 105,0,157,5,6,232,232,232,224,72
30090 DATA 144,232,104,170,104,76,98,228
```

This program first inserts the display list-creating program and the scrolling routine into strings, using the subroutines at 20000 and 30000, respectively. Line 30 simply POKEs some vertical lines into our enlarged display memory so we'll have some information to scroll. Line 40 sets up the new display list, using DLSCROLL$, and passes the address of SCROLL$ to this routine so that it can be inserted into the vertical blank. Since we're not going to do anything except watch the scrolling, line 50 just keeps the real-time program running in a loop while the vertical blank interrupt program (our scrolling routine) continues to do its thing. If you watch this program run for too long, we can't be responsible for your actions — it's hypnotic!

This concludes our review of the display list, display memory, interrupt handling and fine scrolling. You should now be able to write some fairly sophisticated routines in assembly language and use them in your BASIC programs with ease.

CHAPTER NINE
INPUT-OUTPUT ON THE ATARI

THE CENTRAL INPUT-OUTPUT SYSTEM IN ATARI COMPUTERS

In any computer system, the terms **input** and **output** refer to communication between the microprocessor and any external device — a keyboard, the screen editor, a printer, a disk drive, a tape recorder, or other similar peripheral. The ATARI operating system contains the routines for interacting with any of these devices at several levels, but many microcomputers have this ability. The aspect of the ATARI system which makes it unique — and, from a programmer's point of view, so easy to use — is that all external devices are handled identically and are differentiated only by changing minor aspects of the input-output routine.

Input is the passage of information from the outside world, for example, from the keyboard, to the microprocessor. Output is the reverse process, whereby information proceeds from the computer to the outside, to a printer, for example. Throughout the remainder of this book, we will refer to the **C**entral **I**nput-**O**utput system as CIO.

VECTORS IN AN ATARI COMPUTER

We mentioned earlier that the techniques and routines used in this book will work with any ATARI computer because the vectors to the routines in the operating system are guaranteed by ATARI not to change. Located within the operating system of your ATARI is a **jump table**, which contains the addresses of all of the key routines needed for programming in assembly language. The table extends from $E450 through $E47F, and in an ATARI 800 with the B operating system, the table looks like this:

Address	Contains the Instruction
E450	JMP $EDEA
E453	JMP $EDF0
E456	JMP $E4C4
E459	JMP $E959
E45C	JMP $E8ED
E45F	JMP $E7AE
E462	JMP $E905
E465	JMP $E944
E468	JMP $EBF2
E46B	JMP $E6D5
E46E	JMP $E4A6
E471	JMP $F223
E474	JMP $F11B
E477	JMP $F125
E47A	JMP $EFE9
E47D	JMP $EF5D

It's easy to see why this is called a jump table, since it is a table of addresses to which program control will jump when accessed. "Why not jump directly to the given address?" you may ask. In the answer lies the key to writing programs which will run on all ATARI computers. Suppose that rather than accessing $E456, we choose to jump directly to location $E4C4, bypassing the jump table. Everything will work fine, and our program will run. **But,** now suppose that ATARI produces some new computer, the 24800

XLTVB, and the operating system needs to be somewhat altered to accomodate several new features of this magnificent new machine. Our program is in trouble. ATARI never guaranteed that location $E4C4 would stay the same forever; they only guaranteed that the jump table would always point to the right address. That is, if we had accessed $E456 instead of $E4C4, our program would always work, since location $E456 is guaranteed not to change. Let's look at the various vectors in this jump table, with their ATARI equates (the names we'll use for these addresses in any programs we write) and their uses:

Equate	Address	Use
DISKIV	$E450	Disk handler initiation routine
DSKINV	$E453	Disk handler vector
CIOV	$E456	Central Input/Output vector
SIOV	$E459	Serial Input/Output vector
SETVBV	$E45C	Set system timers routine vector
SYSVBV	$E45F	System vertical blank interrupt processing
XITVBV	$E462	Exit from vertical blank processing
SIOINV	$E465	Serial Input/Output initialization
SENDEV	$E468	Serial bus send enable routine
INTINV	$E46B	Interrupt handler routine
CIOINV	$E46E	Central Input/Output initialization
BLKBDV	$E471	Blackboard mode vector to memo pad mode
WARMSV	$E474	Warm start entry (follows SYSTEM RESET)
COLDSV	$E477	Cold start entry point (follows power-up)
RBLOKV	$E47A	Cassette read block routine vector
CSOPIV	$E47D	Cassette open for input vector

We'll be using some of these vectors in the programs we'll write, and some we'll never use, but knowing where they are will help you if you need to make use of them in your own programs. Many are used by the operating system itelf.

To access any of these routines in the operating system, we simply need to JSR to the appropriate address. All of these routines in the operating system are written as subroutines, and therefore end with RTS instructions, which will return control to your program. For instance, to access CIO, we simply need to type

```
JSR CIOV
```

and the job is done. Of course, a considerable amount of setup is required before this call can be made, which we'll be covering shortly; but the actual call to CIO couldn't be simpler.

While we're discussing the available vectors in the operating system, let's briefly cover the RAM and ROM vectors. They are summarized in the following table, with their equates, the information contained in them in the B operating system, and a brief description of their use:

Equate	Address	Points to	Use
CASINI	$0002	varies	Bootable cassette init. vector
DOSINI	$000C	varies	Disk initialization vector
DOSVEC	$000A	varies	Disk software run vector
VDSLST	$0200	$E7B3	DLI NMI vector
VPRCED	$0202	$E7B3	Proceed line IRQ vector **
VINTER	$0204	$E7B3	Interrupt line IRQ vector **
VBREAK	$0206	$E7B3	BRK instruction IRQ vector
VKEYBD	$0208	$FFBE	Keyboard IRQ vector
VSERIN	$020A	$EB11	Serial input ready IRQ vector
VSEROR	$020C	$EA90	Serial output ready IRQ vector
VSEROC	$020E	$EAD1	Serial output done IRQ vector
VTIMR1	$0210	$E7B3	POKEY timer 1 IRQ vector
VTIMR2	$0212	$E7B3	POKEY timer 2 IRQ vector
VTIMR4	$0214	$E7B3	POKEY timer 4 IRQ vector
VIMIRQ	$0216	$E6F6	Vector to IRQ handler
VVBLKI	$0222	$E7D1	Immediate VBI NMI vector
VVBLKD	$0224	$E93E	Deferred VBI blank NMI vector
CDTMA1	$0226	varies	System timer 1 JSR address
CDTMA2	$0228	varies	System timer 2 JSR address
BRKKY	$0236	$E754	BREAK key vector only on "B" OS
RUNVEC	$02E0	varies	Load and go run vector
INIVEC	$02E2	varies	Load & go initialization vector

Those marked with "**" are unused at present. Notice that a number of these vectors point to the same place in the operating system, $E7B3. This is the address of the central interrupt processing routine, which determines the nature of the interrupt and directs program control to the appropriate routines in the operating system to handle that type of interrupt.

These vectors, unlike the ROM vectors, are not arranged in a jump table, so they cannot be accessed by a simple JSR instruction. However, they do point to operating system routines which end in an RTS instruction, so we would like to access them using a JSR instruction. The proper method is to set up a JSR to a location which JMPs indirectly to the above vector. For instance, suppose we want to vector through the DOSINI vector. This is done properly with the following code:

```
40  JSR MYSPOT
45  .
50  .
55  .
60 MYSPOT JMP (DOSINI)
```

Following the JSR to MYSPOT, the RTS in the operating system routine will return control to line 45, at which point your program will resume.

Now that we've seen how to write programs which will work on all ATARI computers, let's discuss the CIO philosophy and learn how to write programs which interact with the real world.

THE INPUT-OUTPUT CONTROL BLOCK (IOCB)

There are two parts to the CIO system in the ATARI. These are the **I**nput-**O**utput **C**ontrol **B**lock, or IOCB, and the **handler table.** Let's discuss these one at a time, and then we'll see how they work together to form an operational CIO system.

The IOCB is a section of memory on page 3 which contains the information that is set up by the programmer to tell the ATARI which device is desired and what information is to be passed. Each IOCB requires 16 bytes of information, and 8 IOCBs are available. Their names and locations are as follows:

Name	Location
IOCB0	$340 to 34F
IOCB1	$350 to 35F
IOCB2	$360 to 36F
IOCB3	$370 to 37F
IOCB4	$380 to 38F
IOCB5	$390 to 39F
IOCB6	$3A0 to 3AF
IOCB7	$3B0 to 3BF

Several of these IOCBs are used by the system as defaults, although as programmers we are free either to use these as the system defaults, or to change them to suit our own purposes. In fact, only 3 of them are normally used by the OS; there is generally no need to redefine them, since we have five others from which to choose. The three used by the OS are as follows:

1. IOCB0, the screen editor. By directing output to IOCB0, we can have information passed to the screen editor. This IOCB also controls the text window in any of the split-screen graphics modes.
2. IOCB6, the screen display for graphics modes higher than zero. This IOCB is used for all graphics commands, like PLOT, DRAWTO, FILL, and others.
3. IOCB7, used to support the LPRINT command of BASIC, which directs output to the printer when this command is used. In practice, much output from BASIC directed to a printer uses one of the other IOCBs, since LPRINTs are not frequently used; more formatting is available if a specific IOCB is OPENed for use with a printer.

As you have probably already recognized, BASIC uses the IOCB numbers (0, 6, and 7) to direct output to these devices, as when printing to a GRAPHICS 1 or 2 screen with this command:

```
PRINT #6;"HELLO"
```

The 16 bytes of the IOCB, and their offsets from the beginning of the IOCB in use, are described below:

Label	Offset	Length	Description
ICHID	0	1	Index into device name table for this IOCB
ICDNO	1	1	Device number
ICCOM	2	1	Command byte: determines action to be taken
ICSTA	3	1	Status returned by device
ICBAL/H	4,5	2	Two-byte buffer address of stored information
ICPTL/H	6,7	2	Address-1 of device's put character routine
ICBLL/H	8,9	2	Buffer length
ICAX1	10	1	First auxiliary byte
ICAX2	11	1	Second auxiliary byte
ICAX3/4	12,13	2	Auxil. bytes 3 & 4 — for BASIC NOTE and POINT
ICAX5	14	1	Fifth auxil. byte — for NOTE and POINT also
ICAX6	15	1	Spare auxilliary byte — unused at present

A SIMPLE I/O EXAMPLE USING AN IOCB

Before getting into the details of the various bytes required for each possible function of an IOCB, an example program will help in understanding their use. Let's take a simple BASIC example and convert it to its assembly language equivalent. The line of BASIC programming we want to duplicate is:

```
CLOSE #4:OPEN #4,6,0,"D:*.*"
```

For now, we need to know that the command byte stored in IC-COM must be $C for the CLOSE command or 3 for the OPEN command, and OPENing the disk directory requires a 6 in ICAX1. Let's look at the program required to OPEN such a file:

```
             0100 ; ******************************
             0110 ; first set up equates
             0120 ; ******************************
0000         0130        *=    $600
0341         0140 ICDNO  =     $0341
0342         0150 ICCOM  =     $0342
0344         0160 ICBAL  =     $0344
```

```
0345          0170 ICBAH  =     $0345
034A          0180 ICAX1  =     $034A
E456          0190 CIOV   =     $E456
              0200 ; *****************************
              0210 ; now CLOSE #4 for insurance
              0220 ; *****************************
0600 A240     0230        LDX  #$40      ;#$40 for IOCB #4
0602 A90C     0240        LDA  #$C       ;CLOSE command byte
0604 9D4203   0250        STA  ICCOM,X   ;X=IOCB #4
0607 2056E4   0260        JSR  CIOV      ;let CIO do the CLOSE
              0270 ; *****************************
              0280 ; now we'll open the directory
              0290 ; *****************************
060A A240     0300        LDX  #$40      ;again, #$40=IOCB4
060C A901     0310        LDA  #1        ;disk drive #1
060E 9D4103   0320        STA  ICDNO,X   ;put drive # here
0611 A903     0330        LDA  #3        ;for OPEN
0613 9D4203   0340        STA  ICCOM,X   ;command byte
0616 A906     0350        LDA  #6        ;for disk directory
0618 9D4A03   0360        STA  ICAX1,X   ;store 6 here
061B A929     0370        LDA  #FILE&255 ;see discussion
061D 9D4403   0380        STA  ICBAL,X   ;low byte buf. addr.
0620 A906     0390        LDA  #FILE/256 ;see discussion
0622 9D4503   0400        STA  ICBAH,X   ;high byte address
0625 2056E4   0410        JSR  CIOV      ;let CIO OPEN it
0628 60       0420        RTS            ;all done
              0430 ; *****************************
              0440 ; now we need the filename
              0450 ; *****************************
0629 44       0460 FILE   .BYTE "D:*.*",$9B
062A 3A
062B 2A
062C 2E
062D 2A
062E 9B
```

In both the CLOSE and OPEN parts of the program, we load the X register with #$40, which will act as the offset into IOCB4. (If we wanted to use IOCB3, we'd simply load the X register with #$30, and so on for all of the other IOCBs.) We then store the command byte $C into ICCOM for that IOCB, and a JSR to CIOV accomplishes the CLOSE for us. It's always a good idea to CLOSE a file before OPENing it, just in case it was already open for some other reason. If the file is already open, you'll get an error on the return from CIO. You can check for an error on any call to the OS by branching to some error-handling routine of your own if after the JSR to CIOV, the minus flag in the processor status register is set. Therefore, we should put a BMI ERROR instruction after the JSR CIOV instruction; but for the purposes of this discussion, we'll assume all is well. You should never make that assumption in your own programs, however.

To OPEN the file, we put a 1 into ICDNO for IOCB4, for disk drive 1, and then we put the command byte 3 into ICCOM for IOCB4 and put a 6 into ICAX1. All we have left to do before the call to CIO is to point the buffer address to the name of the file we want to OPEN. This name is located in line 460, and we've given it the label FILE. The $9B following the file name is the hexadecimal code for a carriage return, which should always follow file or device names, such as S: or P:.

To point the buffer to the file name, we need to break its address into low and high bytes. The low byte is the address AND 255, written #FILE&255. The ANDing with 255 ensures that we get only the low byte of the address. The high byte can be obtained by dividing the address by 256, as we did in line 390. The low and high bytes are stored in ICBAL and ICBAH, respectively, and then a call to CIO in line 410 completes the OPEN command for us.

This simple example demonstrates not only how to open a disk directory, but it also shows exactly how every call to the CIO routine in your ATARI is made. We first set up the appropriate bytes in the IOCB and then simply JSR to CIOV to accomplish the task, whether it is to OPEN a file, READ some information from a disk or tape, or send information to a printer. All of these operations are done using this same sequence of events, which makes input and output in assembly language on your ATARI so simple, once you understand the system. Note that not all of the 16 bytes in the IOCB need to be altered to perform a call to CIO. In fact, we shall see that for some commands, one or two of these bytes are all that are necessary for implementation of the function.

DETAILS OF THE BYTES IN AN IOCB

Now that we've seen how to implement a simple call to the central input-output system of the ATARI computers, we'll review the full spectrum of information which needs to be stored in the various locations of the IOCB in order to implement all possible I/O operations. We'll examine each byte of the IOCB, in the order in which they appear.

The first byte, ICHID, acts as an index into the device table, so you can always tell which device an IOCB is accessing by looking at the first byte. This is set by the OS, and you'll not need to set it for any use. The OS determines this index following the OPEN command and stores the appropriate information here.

ICDNO, the device number, is most often used when more than one disk drive is connected to the system. A different IOCB is used to communicate with each disk drive, and byte 2 of the IOCB distinguishes between the drives in use. If a 1 is stored here, the IOCB will access disk drive 1, and similarly for drives 2 through 4.

The command bytes for the various devices which can be connected to your ATARI computer are as follows:

Command	Byte	Description
Open	3	Open the device for operation
Get record	5	Input a line
Get character	7	Input one or more characters
Put record	9	Output a line
Put character	11	Output one or more characters
Close	12	Close the device
Status	13	Get device status
Draw line	17	Draw a line in GRAPHICS modes
Fill command	18	Fill part of GRAPHICS screen with color
Format disk	254	Format disk

The fourth byte of the IOCB is ICSTA, which is set by the OS following the return from CIO. The status is also set in the Y register upon return from any call to CIO, so either the Y register or ICSTA can be read by your program to determine the success or failure of each I/O operation. Any negative status (value greater than 128 decimal, or $80 hexadecimal) indicates that an error occurred in the I/O operation.

The next 2 bytes of the IOCB act as a pointer to the buffer used for either input or output, and are in the usual 6502 order, low byte first. They are called ICBAL and ICBAH, respectively. A **buffer** is an area of memory which contains the information you wish to output, or into which you want the input information placed. For instance, if you want to send text to a printer, ICBAL and ICBAH are set up to point to the area of memory containing the text to be printed. If you want to read a disk file into memory, these bytes of the IOCB are set up to point to the area of memory where you want the information placed from the disk.

ICPTL and ICPTH act as another 2-byte pointer, but in this case, they point to the address of the put-byte routine of the device, minus 1. Every device which can be opened for output must have a **put-byte routine** written for it, telling the computer how to send information to it. This will be covered more completely when we discuss the handler table.

The next 2 bytes of the IOCB are ICBLL and ICBLH, which contain the length of the I/O buffer, in bytes. As we shall see, there is a special case of I/O in which we set the length of the buffer equal to zero, by setting both ICBLL and ICBLH to zero. In this special case, the information transferred is to or from the accumulator, rather than to or from memory.

Since many devices that can be connected to your ATARI have several possible functions, you must be able to define in the IOCB which function is to be implemented. This is done using the byte in ICAX1, the next byte of the IOCB. The following table lists the various possible bytes for ICAX1. TW refers to a separate text window on the screen, such as that set up by the BASIC command GRAPHICS 3; RE refers to a READ operation enabled from the screen; and RD means that such a READ is not allowed, or disabled.

Device	ICAX1 Byte	Function
Screen editor	8	Output to the screen
	12	Input from the keyboard and output to screen
	13	Forced screen input and output
Screen display	8	Screen is cleared; no TW; RD
	12	Screen is cleared; no TW; RE
	24	Screen is cleared; TW; RD
	28	Screen is cleared; TW; RE
	40	Screen is not cleared; no TW; RD
	44	Screen is not cleared; no TW; RE
	56	Screen is not cleared; TW; RD
	60	Screen is not cleared; TW; RE
Keyboard	4	Read — note:no output is possible
Printer	8	Write — note:no input is possible
Tape recorder	4	Read
	8	Write
RS-232 port	5	Concurrent read
	8	Block write
	9	Concurrent write
	13	Concurrent read and write
Disk drive	4	Read
	6	Read disk directory
	8	Write new file
	9	Write — append
	12	Read and write — update mode

The last byte of the IOCB we will discuss here is ICAX2, the second auxiliary byte. ICAX2 is used in only a few special cases; otherwise, it is set to zero. When using the cassette recorder, if a value of 128 is stored in ICAX2, the short interrecord gaps, the silent spaces between sections of information on the tape, are used, which will allow faster loads of a tape written in this manner. A value of zero in ICAX2 will produce the normal, longer interrecord gaps.

Using the ATARI 820 printer, storing a value of 83 in ICAX2 will cause the printer to print sideways instead of in its normal mode. Furthermore, values of 70 or 87 in ICAX2 will produce normal or double-width characters on this printer.

Finally, graphics modes 0 to 11 are specified in the OPEN command by placing the number of the desired mode in ICAX2. In combination with the values described for ICAX1 above, ICAX2 gives the assembly language programmer complete control over the graphics mode, text window, screen clear, and read-write functions of the screen. We'll learn more about this in Chapter 10.

THE HANDLER TABLE

Now that we've covered the various parts of an IOCB, we'll briefly describe the handler table and how it works with the IOCBs to form the I/O system with CIO. Then we'll look at a number of examples which will show how to use this information to perform many different types of I/O from assembly language. The simplest way to examine the handler table is to view it as a short assembly language program, such as this:

```
0100 PRINTV = $E430
0110 CASETV = $E440
0120 EDITRV = $E400
0130 SCRENV = $E410
0140 KEYBDV = $E420
0150 ; ******************************
0160 ; Origin of HATABS = $031A
0170 ; ******************************
0180  *= $031A
0190 .BYTE "P"      ;printer
0200 .WORD PRINTV ;   vector
0210 .BYTE "C"      ;cassette recorder
0220 .WORD CASETV  ;   vector
0230 .BYTE "E"      ;editor
0240 .WORD EDITRV  ;   vector
0250 .BYTE "S"      ;screen
0260 .WORD SCRENV  ;   vector
0270 .BYTE "K"      ;keyboard
0280 .WORD KEYBDV  ;   vector
0290 .BYTE 0        ;free entry #1(DOS)
0300 .WORD 0,0
0310 .BYTE 0        ;free entry #2(850 interface)
```

```
0320 .WORD 0,0
0330 .BYTE 0        ;free entry #3
0340 .WORD 0,0
0350 .BYTE 0        ;free entry #4
0360 .BYTE 0,0
0370 .BYTE 0        ;free entry #5
0380 .WORD 0,0
0390 .BYTE 0        ;free entry #6
0400 .WORD 0,0
0410 .BYTE 0        ;free entry #7
0420 .WORD 0,0
```

Each entry in the handler table consists of the first letter of the specified device, followed by the vector which points to the location in memory of the information needed to deal with that device. As you can see, there are seven free places left in the handler table, so the programmer is free to add whatever devices are necessary for any purpose, and they'll be treated just like the devices already specified. One other very important point about the handler table should be noted here. Whenever the OS looks into the handler table to find out where in memory it needs to look to take care of a particular device, it reads the table from the bottom up! This is intentional and allows you to insert your own printer handler near the bottom of the table. As the table is searched, your vector will be found first, and it is the one that will be used. Therefore, you can write your own printer-handling routines and substitute these for the normal routines easily, simply by placing another P: in one of the lower free entries and following it by the 2-byte address vector pointing to your new handling routines.

Let's briefly look at a typical **handler entry point table,** which is the table to which the entry in the handler table points. For example, the vector PRINTV, used above, points to a second table, the printer handler entry point table. In fact, all of the above vectors point to their respective handler entry point tables, and all of these tables are arranged identically. They contain the addresses minus 1 of the routines used for the following functions, in the following order:

OPEN the device routine
CLOSE the device routine

READ routine
WRITE routine
STATUS of the device routine
SPECIAL functions, where implemented

The handler entry point table is always terminated by a 3-byte JMP instruction, which points to the initialization routine for that device. *Remember:* the addresses found in the handler entry point table *do not* point to the OPEN and CLOSE routines, but rather, they point to the address 1 byte lower in memory than the beginning of each of these routines. It is obviously very important to remember this when you are constructing your own handler entry point table!

A SIMPLE I/O ROUTINE

Let's see how we can use CIO for a simple function — writing to the screen. We know that in BASIC, if we want to write a line of text to the screen, all that's required is a single line of code like this:

```
PRINT "A SUCCESSFUL WRITE!"
```

In assembly language, it's also fairly simple to print to the screen, now that we understand the use of the IOCB and CIO. Just to review, we don't have to open the screen as a device if we don't want to, since IOCB0 is already allocated by the OS for the screen. Therefore, we can load the X register with zero and use that as an offset into the IOCB. Alternatively, we can just use absolute addressing, since we'll be using the first IOCB. In the example below, we'll use the X register loaded with zero, just so we become familiar with the normal procedure for inserting the required information into the IOCB. Here's the routine to write the line to the screen:

```
               ; *****************************
               ; CIO equates
               ; *****************************
0340           ICHID   =     $0340
0341           ICDNO   =     $0341
```

```
0342             ICCOM  =     $0342
0343             ICSTA  =     $0343
0344             ICBAL  =     $0344
0345             ICBAH  =     $0345
0346             ICPTL  =     $0346
0347             ICPTH  =     $0347
0348             ICBLL  =     $0348
0349             ICBLH  =     $0349
034A             ICAX1  =     $034A
034B             ICAX2  =     $034B
E456             CIOV   =     $E456
0000                    *=    $600
                 ; ******************************
                 ; Now we load in required data
                 ; ******************************
0600 A200               LDX  #0          ;Since it's IOCB0
0602 A909               LDA  #9          ;For put record
0604 9D4203             STA  ICCOM,X     ;Command byte
0607 A91F               LDA  #MSG&255    ;Low byte of MSG
0609 9D4403             STA  ICBAL,X     ;  into ICBAL
060C A906               LDA  #MSG/256    ;High byte of MSG
060E 9D4503             STA  ICBAH,X     ;  into ICBAH
0611 A900               LDA  #0          ;Length of MSG
0613 9D4903             STA  ICBLH,X     ;  high byte
0616 A9FF               LDA  #$FF        ;>length of MSG
0618 9D4803             STA  ICBLL,X     ;  see discussion
                 ; ******************************
                 ; Now put it to the screen
                 ; ******************************
061B 2056E4             JSR  CIOV
061E 60                 RTS
                 ; ******************************
                 ; The message itself
                 ; ******************************
061F 41          MSG    .BYTE "A SUCCESSFUL WRITE!",$9B
0620 20
0621 53
0622 55
0623 43
0624 43
0625 45
0626 53
```

```
0627 53
0628 46
0629 55
062A 4C
062B 20
062C 57
062D 52
062E 49
062F 54
0630 45
0631 21
0632 9B
```

Of course, writing to the screen is so simple in BASIC that there would be no reason to write this program as a subroutine for BASIC, so it doesn't contain the usual PLA instruction. Since you will need to print to the screen to debug assembly language programs, this routine may become one of your most frequently used programs.

In order to test this program once you have entered it, simply type ASM to assemble it, and when the assembly is complete, type BUG to enter the DEBUG mode of the Assembler/Editor cartridge. Then type G600 to begin execution at address $600. If the program has been typed correctly, the phrase A SUCCESSFUL WRITE! should appear, followed by the printing to the screen of the 6502 registers. These are printed following every routine that uses the cartridge. This same procedure should be used to test each of the routines given in this book. If problems arise, check your typing.

PLEASE NOTE!!! SAVE YOUR PROGRAMS **BEFORE** YOU TRY TO RUN THEM!!! Then if they fail, or you have a computer crash, you won't have to retype the entire program

In this program, we write the entire message to the screen by using the put-record command, storing a 9 into ICCOM. The address of the message we want to display on the screen is then stored into ICBAL and ICBAH, as before. We store a zero into the high byte of the length of the message, but $FF into the low byte.

Why $FF when the message is only 20 bytes long? When CIO is used in the put-record mode, the record is output byte by byte, until either the length of the buffer, set in ICBLL and ICBLH, has been exceeded or until a RETURN is encountered in the record being output. Note that the message which was set up in line 490 terminates with a byte of $9B, which is a RETURN. Therefore, the message will be sent to the screen, then a carriage return will be sent to the screen, and then the routine will terminate. We set the length of the record intentionally longer than the real message, because we want the $9B in the message itself to terminate the output. This way, we can't make a mistake and unintentionally cut the message short by setting ICBLL or ICBLH smaller than we intended.

It is important to note that we didn't set all of the bytes in the IOCB. In fact, we only needed to set the bytes that our particular routine used. As you'll see below, this is always the case with the central ATARI routines. The actual output to the screen is accomplished by the call to the central I/O routine in line 440, and the RTS in the next line returns control to the Assembler/Editor cartridge. If this were part of a larger assembly language program, the rest of the program would continue from line 450 without the RTS.

OTHER FORMS OF THE I/O ROUTINE

Let's look at another way to write a message to the screen, using the CIO system. Instead of loading ICCOM with 9, for put record, we can load it with 11, for put bytes. The other bytes of the IOCB are set as above, except for ICBLL, which is set to the exact length of the message. When counting the bytes in the message, don't forget to include the byte for the $9B, the RETURN. The program will then look like this:

```
                0100 ; ******************************
                0110 ; CIO equates
                0120 ; ******************************
0340            0130 ICHID  =     $0340
0341            0140 ICDNO  =     $0341
```

```
0342          0150 ICCOM  =     $0342
0343          0160 ICSTA  =     $0343
0344          0170 ICBAL  =     $0344
0345          0180 ICBAH  =     $0345
0346          0190 ICPTL  =     $0346
0347          0200 ICPTH  =     $0347
0348          0210 ICBLL  =     $0348
0349          0220 ICBLH  =     $0349
034A          0230 ICAX1  =     $034A
034B          0240 ICAX2  =     $034B
E456          0250 CIOV   =     $E456
0000          0260        *=    $600
              0270 ; ******************************
              0280 ; Now we load in required data
              0290 ; ******************************
0600 A200     0300        LDX  #0         ;Since it's IOCB0
0602 A90B     0310        LDA  #11        ;For put bytes
0604 9D4203   0320        STA  ICCOM,X    ;Command byte
0607 A91F     0330        LDA  #MSG&255   ;Low byte of MSG
0609 9D4403   0340        STA  ICBAL,X    ;  into ICBAL
060C A906     0350        LDA  #MSG/256   ;High byte of MSG
060E 9D4503   0360        STA  ICBAH,X    ;  into ICBAH
0611 A900     0370        LDA  #0         ;Length of MSG
0613 9D4903   0380        STA  ICBLH,X    ;  high byte
0616 A914     0390        LDA  #20        ;length of MSG
0618 9D4803   0400        STA  ICBLL,X    ;  low byte
              0410 ; ******************************
              0420 ; Now put it to the screen
              0430 ; ******************************
061B 2056E4   0440        JSR  CIOV
061E 60       0450        RTS
              0460 ; ******************************
              0470 ; The message itself
              0480 ; ******************************
061F 41       0490 MSG    .BYTE "A SUCCESSFUL WRITE!",$9B
0620 20
0621 53
0622 55
0623 43
0624 43
0625 45
0626 53
```

```
0627 53
0628 46
0629 55
062A 4C
062B 20
062C 57
062D 52
062E 49
062F 54
0630 45
0631 21
0632 9B
```

This program accomplishes exactly the same end that the previous routine does, but in a different way. Both of these programs write a message to the screen that ends in a carriage return. There is a special case of writing to the screen in which we do not want the text to be followed by a return, such as when we are prompting for input, or when we would like to format the screen in a particular way. In BASIC, this instruction simply is a PRINT statement followed by a semicolon, which inhibits the normal carriage return following a PRINT command. If, for instance, we want to print a > symbol to the screen to prompt the user for input, but we want the cursor to remain on the same line as the symbol, in BASIC we could write the following line to accomplish this task:

```
PRINT ">";
```

In assembly language programming, the code is as follows:

```
                0100 ; ******************************
                0110 ; CIO equates
                0120 ; ******************************
0340            0130 ICHID  =     $0340
0341            0140 ICDNO  =     $0341
0342            0150 ICCOM  =     $0342
0343            0160 ICSTA  =     $0343
0344            0170 ICBAL  =     $0344
0345            0180 ICBAH  =     $0345
```

```
0346          0190 ICPTL  =    $0346
0347          0200 ICPTH  =    $0347
0348          0210 ICBLL  =    $0348
0349          0220 ICBLH  =    $0349
034A          0230 ICAX1  =    $034A
034B          0240 ICAX2  =    $034B
E456          0250 CIOV   =    $E456
0000          0260         *=  $600
              0270 ; ******************************
              0280 ; Now we load in required data
              0290 ; for special 1-character case
              0300 ; ******************************
0600 A200     0310        LDX  #0         ;Since it's IOCB0
0602 A90B     0320        LDA  #11        ;For put bytes
0604 9D4203   0330        STA  ICCOM,X    ;Command byte
0607 A900     0340        LDA  #0         ;Length of MSG
0609 9D4903   0350        STA  ICBLH,X    ;  high byte
060C A900     0360        LDA  #0         ;length of MSG
060E 9D4803   0370        STA  ICBLL,X    ;  low byte
              0380 ; ******************************
              0390 ; Now put it to the screen
              0400 ; ******************************
0611 A93E     0410        LDA  #62        ;For ">"
0613 2056E4   0420        JSR  CIOV
0616 60       0430        RTS
```

If we set the length of the buffer equal to zero (by setting both the high and low bytes, ICBLL and ICBLH, to zero), then the character contained in the accumulator when CIOV is accessed will be printed to the output device without a following carriage return. This applies to all devices, including disk drives, tape recorders, printers and the screen, and it points out a very important feature of the ATARI computers: input and output are largely device-independent. That is, the OS treats all devices similarly, so we don't have to learn how to write a message to the screen, then learn a different way to send information to the printer, and learn still another method for passing information to the disk drive. The method is identical, once the IOCB has been opened for the device. To demonstrate this, we'll look at a routine to send the same message to a printer.

OUTPUT TO A PRINTER

First we'll close IOCB2, just to be on the safe side; then we'll open the printer as a device using IOCB2; and then we'll send our message.

```
                ; *****************************
                ; CIO equates
                ; *****************************
0340            ICHID  =     $0340
0341            ICDNO  =     $0341
0342            ICCOM  =     $0342
0343            ICSTA  =     $0343
0344            ICBAL  =     $0344
0345            ICBAH  =     $0345
0346            ICPTL  =     $0346
0347            ICPTH  =     $0347
0348            ICBLL  =     $0348
0349            ICBLH  =     $0349
034A            ICAX1  =     $034A
034B            ICAX2  =     $034B
E456            CIOV   =     $E456
0000                  *=     $600
                ; *****************************
                ; First, close and open IOCB2
                ; *****************************
0600 A220              LDX  #$20       ;for IOCB2
0602 A90C              LDA  #12        ;close command
0604 9D4203            STA  ICCOM,X    ;   into ICCOM
0607 2056E4            JSR  CIOV       ;do the CLOSE
060A A220              LDX  #$20       ;IOCB2 again
060C A903              LDA  #3         ;open file
060E 9D4203            STA  ICCOM,X    ;  is the command
0611 A908              LDA  #8         ;output
0613 9D4A03            STA  ICAX1,X    ;  open for output
0616 A94C              LDA  #NAM&255   ;low byte of device
0618 9D4403            STA  ICBAL,X    ;points to "P:"
061B A906              LDA  #NAM/256   ;high byte
061D 9D4503            STA  ICBAH,X
0620 A900              LDA  #0
0622 9D4903            STA  ICBLH,X    ;high byte length
```

```
0625 A9FF   0450        LDA  #$FF
0627 9D4803 0460        STA  ICBLL,X    ;>low byte length
062A 2056E4 0470        JSR  CIOV       ;do the OPEN
            0480 ; ******************************
            0490 ; Now we'll print the message
            0500 ; ******************************
062D A220   0510        LDX  #$20       ;by using IOCB2
062F A909   0520        LDA  #9         ;put record
0631 9D4203 0530        STA  ICCOM,X    ;   command
0634 A94F   0540        LDA  #MSG&255   ;address of MSG
0636 9D4403 0550        STA  ICBAL,X    ;  low byte
0639 A906   0560        LDA  #MSG/256   ;address of MSG
063B 9D4503 0570        STA  ICBAH,X    ;  high byte
063E A900   0580        LDA  #0         ;length of MSG
0640 9D4903 0590        STA  ICBLH,X    ;  high byte
0643 A9FF   0600        LDA  #$FF       ;>length of MSG
0645 9D4803 0610        STA  ICBLL,X    ;  low byte
0648 2056E4 0620        JSR  CIOV       ;put out the line
064B 60     0630        RTS             ;end of routine
064C 50     0640 NAM    .BYTE "P:",$9B
064D 3A
064E 9B
064F 41     0650 MSG    .BYTE "A SUCCESSFUL WRITE!",$9B
0650 20
0651 53
0652 55
0653 43
0654 43
0655 45
0656 53
0657 53
0658 46
0659 55
065A 4C
065B 20
065C 57
065D 52
065E 49
065F 54
0660 45
0661 21
0662 9B
```

Of course, if you are using an ATARI printer and want to print in expanded print, you'll have to set ICAX1 and ICAX2 before the final call to CIOV, but that's trivial. Note that we have not CLOSEd IOCB2 following the printing of our message, so if we want to print anything else, we can simply send it through IOCB2 without needing to OPEN it again. Of course, that also means that now we can't use IOCB2 for anything else, such as disk access. If we need to access the disk, we can use one of the other IOCBs, or we can CLOSE IOCB2 first and then reOPEN it for our disk operation.

Note that if we want the printhead to stop after printing a single character, without a trailing carriage return, we can use the special case of zero-length buffer, exactly as we did above for the screen.

OUTPUT TO A DISK

In order to show the versatility of the ATARI central I/O routines, we won't even give the program here to write the same line to a disk file. The method will be described, and you'll be able to send information to your disk on the first try, all by yourself! The only change needed in the program given above for the printer is this: to use the disk drive, the NAMe of the device is the disk file you wish to OPEN. Therefore, the program is identical to that given above, but line 640 should read something like:

```
640 NAM .BYTE "D1:MYFILE.1",$9B
```

That's all there is to it. You can see the beauty of using identical CIO routines for all devices, and you should now be able to output information to any device of your choosing in assembly language.

INPUT USING CIOV

The method for input from a device to your ATARI computer is exactly the same as that for output, but the device must be

OPENed for input. We could, for instance, retrieve the above message from our disk file "D1:MYFILE.1" by OPENing this file for input, using a 3 in ICCOM and a 4 in ICAX1, and pointing ICBAL and ICBAH to the location in memory to which we want the message transferred. For instance, if we want the message to begin at memory location $680, we would set ICBAL to #$80 and ICBAH to #6; after the call to CIOV, memory locations $680 through $694 will contain the bytes of the message, which can then be examined by the remainder of our program.

The ease and simplicity of this I/O on an ATARI computer should not be underestimated. Learning each device is a separate chore with many other microcomputers; input and output may use different routines, each with their own peculiarities. The central I/O philosophy used in the ATARI greatly simplifies this process for us. Now you can use this system to greatly enhance your assembly language programming abilities.

One final note on the I/O routines: if we OPENed one IOCB for input from a file on the disk drive and a second IOCB for output to a printer or to the screen, it would be an trivial task to transfer information very quickly from one device to another by pointing to the same buffer for both IOCBs. Printing hard copy from and copying memory to a disk file is simple. We can even transfer information from the disk drive to the screen, or to a tape recorder.

NON-CIO INPUT AND OUTPUT

THREE DIFFERENT I/O SYSTEMS

Besides CIO, there are two other methods for using the disk drive as an input-output device; both reside in the OS. They use vectors called DSKINV and SIOV, at $E453 and $E459, respectively.

The three methods of disk I/O can be viewed as an onion, with multiple layers of control. The outer layer, which does most of the work for you, is the CIO system; the middle layer, which does some of the work for you, is the DSKINV system; and the inner layer, in which the programmer does all of the work, is the SIOV system. In fact, SIOV, the serial input-output vector, is used for all communications which take place over the **serial bus**, the 13-pronged connector on the side of your ATARI computer. Even the CIO system performs the actual input-output operations by calling SIO, after using the information in the IOCB to set up everything for SIO. DSKINV, about which we will learn more shortly, also calls SIO to perform the actual I/O.

DISK FILE TYPES

A floppy disk for your ATARI disk drive contains 40 concentric tracks, somewhat like a phonograph record. On a record, however, the tracks are actually one continuous spiral, whereas on a floppy disk, each track is a separate circle. Each track is divided into 18 sectors. To envision this, imagine cutting the disk like a pizza, with 18 equal slices. Then cut the pizza into 40 concentric circles, like a bullseye with 40 different colored rings. Each piece of the pizza is one **sector**. We'll have 18 X 40, or 720 sectors. On each of the sectors, the ATARI can store 128 bytes of information.

In the process of **formatting** a disk, not only are the 720 sectors created, but a Volume Table Of Contents (VTOC) and a disk directory are also created. The disk directory acts just like the table of contents of a book, listing each file (chapter) contained on the disk and the sector number where that file can be found (page). The VTOC keeps track of which sectors have already been filled and which remain empty, so that when we save a new file onto a partially full disk, we won't write over information already stored in another file. When we delete a file, its sectors are freed in the VTOC so that they can be used again. Note that when a file is deleted, only 1 byte of the file is actually changed — the status, or flag, byte. The first five bytes of the disk directory entry for each file are:

1. The status byte, which contains the status of the file. Each of 4 bits of the status byte are used to store specific information about that file:

Bit 0 set if file is open for output
Bit 5 set if file is locked
Bit 6 set if file is in use
Bit 7 set if file was deleted

2, 3. The length of the file, in sectors, in the usual low byte-high byte order.

4, 5. The number of the first sector of the file in low-high order.

If you have any of the many disk utility programs available, you can actually retrieve a deleted file, simply by changing the status byte from $80 to $40. However, if you write any information to the disk prior to trying this procedure, it will not work. The VTOC is changed when a file is deleted, freeing the sectors for use; if you've written to the disk, you'll find that some of the sectors previously used for the file you want to retrieve have been overwritten by the new information.

For the purpose of this discussion, we'll describe two different file types used by the ATARI computers. The first, and by far the most common, is the linked file, such as that created by this BASIC instruction:

```
SAVE "D:GAME"
```

First, the disk directory is searched. Since a maximum of 64 files can be contained on a single disk, this check ensures that there is room in the disk directory for another file, called GAME. If, when checking the directory, a file called GAME is encountered, it is deleted (unless it is locked) and the new file replaces the old one. Assuming that this is the first GAME file to be SAVEd and that there is room in the disk directory, the first 125 bytes of the new file GAME are written to the first sector which the VTOC says is available. Note that only 125 bytes of the file are written, even though each sector can hold 128 bytes. This leaves room for the 3 bytes added by CIO, which lead to the name **linked file** for this type of file.

These 3 bytes contain the following information:

Byte Number		
125	126	127
765432 10	76543210	7 6543210
file #	forward link	S byte ct

The high 6 bits of byte 125 are the file number, taken from the number of the file in the disk directory. For instance, if GAME is the fourth file listed in the directory, then the file number contained in the high 6 bits of byte 125 of every sector of GAME is 3, since the numbering starts with file 0. This number is checked when reading this file to ensure that each sector really belongs to the file GAME. If, when reading a file, a sector is encountered with a different file number, an error message will be displayed on your screen. This usually means that things have really been messed up on your disk; trying to fix such a file is a major undertaking.

The low order 2 bits of byte 125 are combined with byte 126 to produce a 10-bit number containing the number of the next sector of the file. Therefore, after the first sector is read, the next sector to be read can be determined from this forward link, and so on, until the whole file is read. That is why we call this a linked file. The last sector of each file contains 00 as a forward link, so we can determine when the entire file has been read.

Byte 127 of each sector of a linked file contains the number of bytes stored into that sector. In every sector except the last of each file, this will equal 125. If fewer than 125 bytes are contained in the sector, the high bit of byte 127, the S bit, will be set, denoting a Short sector of less than 125 bytes.

The second major type of disk file is called the **sequential file**, and is much simpler in structure. It uses neither the disk directory nor the VTOC, and uses all 128 bytes of each sector for storage. The sectors are read from such a file sequentially: sector 3 is read after sector 2, which was read after sector 1. The first sector of such a file contains the load address (where in memory to load this file) and the start address (where to begin execution of the program once the load is complete). This type of file is usually found on commercially available games. If you attempt to look at the disk directory of such a disk, you'll see only garbage, since no directory was ever set up for that disk.

USE OF THE DIFFERENT I/O SYSTEMS

CIO is generally used to read a linked file, such as a BASIC program, or, for that matter, the source code for an assembly language program. However, when you want to read a specific sector from the disk, generally DSKINV or SIO is used. Let's examine how we would accomplish these tasks using the three different types of I/O calls.

First, we'll open a disk file and read it into memory. The segment of the program that opens a file is very similar to the program above which opened the disk directory:

```
              ; *****************************
              ; CIO equates
              ; *****************************
0340          ICHID  =    $0340
0341          ICDNO  =    $0341
0342          ICCOM  =    $0342
0343          ICSTA  =    $0343
0344          ICBAL  =    $0344
0345          ICBAH  =    $0345
0346          ICPTL  =    $0346
0347          ICPTH  =    $0347
0348          ICBLL  =    $0348
0349          ICBLH  =    $0349
034A          ICAX1  =    $034A
034B          ICAX2  =    $034B
E456          CIOV   =    $E456
0000                *=    $600
              ; *****************************
              ; Open a file called OBJECT.COD
              ; *****************************
0600 A220            LDX  #$20        ;use IOCB2
0602 A90C            LDA  #12         ;to close IOCB
0604 9D4203          STA  ICCOM,X     ;command byte
0607 2056E4          JSR  CIOV        ;do the close
              ; *****************************
060A A220            LDX  #$20        ;use IOCB2 again
060C A903            LDA  #3          ;open command
060E 9D4203          STA  ICCOM,X     ;command byte
0611 A904            LDA  #4          ;open for read
```

```
0613 9D4A03 0390        STA  ICAX1,X    ;  into ICAX1
0616 A900   0400        LDA  #0         ;0 into ICAX2 is
0618 9D4B03 0410        STA  ICAX2,X    ;just for insurance
061B A94F   0420        LDA  #NAME&255 ;low byte of file
061D 9D4403 0430        STA  ICBAL,X    ;  name address
0620 A906   0440        LDA  #NAME/256 ;high byte - file
0622 9D4503 0450        STA  ICBAH,X    ;  name address
0625 2056E4 0460        JSR  CIOV       ;open the file
            0470 ; ******************************
0628 A220   0480        LDX  #$20       ;IOCB2
062A A900   0490        LDA  #0
062C 9D4403 0500        STA  ICBAL,X    ;low byte-address
062F A950   0510        LDA  #$50       ;high byte-address
0631 9D4503 0520        STA  ICBAH,X    ;  is then $5000
0634 A9FF   0530        LDA  #$FF       ;make buffer length
0636 9D4803 0540        STA  ICBLL,X    ;  very long so the
0639 9D4903 0550        STA  ICBLH,X    ;  whole file loads
063C A905   0560        LDA  #5         ;get record
063E 9D4203 0570        STA  ICCOM,X    ;command byte
0641 2056E4 0580        JSR  CIOV       ;read the whole file
            0590 ; ******************************
0644 A220   0600        LDX  #$20       ;IOCB2
0646 A90C   0610        LDA  #12        ;to close IOCB
0648 9D4203 0620        STA  ICCOM,X    ;command byte
064B 2056E4 0630        JSR  CIOV       ;do the close
064E 60     0640        RTS             ;end of the routine
            0650 ; ******************************
064F 44     0660 NAME   .BYTE "D1:OBJECT.COD",$9B
0650 31
0651 3A
0652 4F
0653 42
0654 4A
0655 45
0656 43
0657 54
0658 2E
0659 43
065A 4F
065B 44
065C 9B
```

This program makes use of a trick to load the entire file in one operation. In lines 530 to 550, we set the length of the buffer to $FFFF, or 65,535 bytes. The CIO routine then will load the entire file, stopping either when 65,535 bytes have been loaded (an impossibility) or when an end-of-line byte is encountered. Therefore, if our file contains any end-of-line bytes ($9B), the load will terminate, and we won't load the entire file. How can we get around this problem?

Since we probably won't know for sure whether the file to be loaded will contain any $9B bytes, we should play it safe and use a method which will load any file. To do this, we load one sector (128 bytes) at a time, continuing until an error condition is achieved, which will occur at the end of the file. Using CIO, we know when an error occurs, since we will return from the call to CIO with the negative flag set. Let's take a look at the program to perform this type of load using CIO:

```
                0100 ; *****************************
                0110 ; CIO equates
                0120 ; *****************************
0340            0130 ICHID  =     $0340
0341            0140 ICDNO  =     $0341
0342            0150 ICCOM  =     $0342
0343            0160 ICSTA  =     $0343
0344            0170 ICBAL  =     $0344
0345            0180 ICBAH  =     $0345
0346            0190 ICPTL  =     $0346
0347            0200 ICPTH  =     $0347
0348            0210 ICBLL  =     $0348
0349            0220 ICBLH  =     $0349
034A            0230 ICAX1  =     $034A
034B            0240 ICAX2  =     $034B
E456            0250 CIOV   =     $E456
0000            0260        *=    $600
                0270 ; *****************************
                0280 ; Open a file called OBJECT.COD
                0290 ; *****************************
0600 A220       0300        LDX   #$20       ;use IOCB2
0602 A90C       0310        LDA   #12        ;to close IOCB
```

```
0604 9D4203 0320          STA  ICCOM,X    ;command byte
0607 2056E4 0330          JSR  CIOV       ;do the close
            0340 ; *****************************
060A A220   0350          LDX  #$20       ;use IOCB2 again
060C A903   0360          LDA  #3         ;open command
060E 9D4203 0370          STA  ICCOM,X    ;command byte
0611 A904   0380          LDA  #4         ;open for read
0613 9D4A03 0390          STA  ICAX1,X    ;  into ICAX1
0616 A900   0400          LDA  #0         ;0 into ICAX2 is
0618 9D4B03 0410          STA  ICAX2,X    ;just for insurance
061B A969   0420          LDA  #NAME&255  ;low byte of file
061D 9D4403 0430          STA  ICBAL,X    ;  name address
0620 A906   0440          LDA  #NAME/256  ;high byte - file
0622 9D4503 0450          STA  ICBAH,X    ;  name address
0625 2056E4 0460          JSR  CIOV       ;open the file
            0470 ; *****************************
0628 A220   0480          LDX  #$20       ;IOCB2
062A A900   0490          LDA  #0
062C 9D4803 0500          STA  ICBLL,X    ;lo buffer length
062F A980   0510          LDA  #$80       ;to load one sector
0631 9D4903 0520          STA  ICBLH,X    ;  at a time
0634 A950   0530          LDA  #$50       ;high byte of
0636 9D4503 0540          STA  ICBAH,X    ;  buffer address
0639 A905   0550          LDA  #5         ;get record
063B 9D4203 0560          STA  ICCOM,X    ;command byte
063E A220   0570 LOOP     LDX  #$20       ;for when looping
0640 A900   0580          LDA  #0         ;lo byte of buffer
0642 9D4403 0590          STA  ICBAL,X    ;  address @ start
0645 2056E4 0600          JSR  CIOV       ;read 1st sector
0648 3014   0610          BMI  FIN        ;if done, to FIN
064A A220   0620          LDX  #$20       ;IOCB2
064C A980   0630          LDA  #$80       ;move up 128 bytes
064E 9D4403 0640          STA  ICBAL,X    ;  for buffer
0651 2056E4 0650          JSR  CIOV       ;read next sector
0654 3008   0660          BMI  FIN        ;if done, to FIN
0656 A220   0670          LDX  #$20       ;IOCB2
0658 FE4503 0680          INC  ICBAH,X    ;raise buffer again
065B 4C3E06 0690          JMP  LOOP       ;not done-read more
            0700 ; *****************************
065E A220   0710 FIN      LDX  #$20       ;IOCB2
0660 A90C   0720          LDA  #12        ;to close IOCB
0662 9D4203 0730          STA  ICCOM,X    ;command byte
```

```
0665 2056E4 0740        JSR  CIOV       ;do the close
0668 60     0750        RTS             ;end of the routine
            0760 ; ****************************
0669 44     0770 NAME   .BYTE "D1:OBJECT.COD",$9B
066A 31
066B 3A
066C 4F
066D 42
066E 4A
066F 45
0670 43
0671 54
0672 2E
0673 43
0674 4F
0675 44
0676 9B
```

We continue to loop until we encounter an error on I/O, at which time we branch to FIN to close the file and finish the routine. You must be careful that no errors other than the end-of-file error occur, since this program will branch to FIN on any error. It would, of course, be fairly easy to write a routine to first determine the error code returned in the Y register after the call to CIOV and then take appropriate action depending on the error code. Note that in this routine, we have to take care of some of the housekeeping for loading the file, such as incrementing the buffer address in lines 630, 640, and 680; we didn't have to worry about this in the first example. We also have to build in routines that we didn't formerly need to determine when we are done loading.

LOADING USING THE RESIDENT DISK HANDLER

In order to utilize the resident disk handler, the programmer must set up a **D**evice **C**ontrol **B**lock (DCB), which is exactly analogous to the IOCB we need to set up when we use CIO. The equates for this DCB are as follows:

```
0100 ; ******************************
0110 ; SIO equates
0120 ; ******************************
0130 DDEVIC = $0300 ;serial bus I.D.
0140 DUNIT = $0301  ;device number
0150 DCOMND = $0302 ;command byte
0160 DSTATS = $0303 ;status byte
0170 DBUFLO = $0304 ;lo buffer addr.
0180 DBUFHI = $0305 ;hi buffer addr.
0190 DTIMLO = $0306 ;disk timeout
0210 DBYTLO = $0308 ;lo byte count
0220 DBYTHI = $0309 ;hi byte count
0230 DAUX1 = $030A  ;auxiliary #1
0240 DAUX2 = $030B  ;auxiliary #2
0250 SIOV  = $E459
0260 DSKINV = $E453
```

The third byte of both the IOCB and DCB is the command byte, although the command bytes themselves are different in the two systems, and the fifth and sixth bytes of both systems are the buffer address. Only the following 5 command bytes are allowed by the resident disk handler:

$21 format a disk
$50 write a sector
$52 read a sector
$53 status request
$57 write a sector with write-verify

It is therefore apparent that the resident disk handler is a more limited but a far simpler system than CIO. Let's look at how we can use the DCB and the resident disk handler, through DSKINV, to read information from the disk. Of course, we will not be reading regular DOS files using this system; they are linked files, and the resident disk handler is not designed to handle linked files, but rather sequential ones. Let's therefore assume that we want to read sectors $20 through $60, inclusive, rather than some disk file. The program to do this using DSKINV follows:

```
                0100 ; *****************************
                0110 ; SIO equates
                0120 ; *****************************
0300            0130 DDEVIC =      $0300       ;serial bus I.D.
0301            0140 DUNIT  =      $0301       ;device number
0302            0150 DCOMND =      $0302       ;command byte
0303            0160 DSTATS =      $0303       ;status byte
0304            0170 DBUFLO =      $0304       ;lo buffer addr.
0305            0180 DBUFHI =      $0305       ;hi buffer addr.
0306            0190 DTIMLO =      $0306       ;disk timeout
0308            0200 DBYTLO =      $0308       ;lo byte count
0309            0210 DBYTHI =      $0309       ;hi byte count
030A            0220 DAUX1  =      $030A       ;auxiliary #1
030B            0230 DAUX2  =      $030B       ;auxiliary #2
E459            0240 SIOV   =      $E459
E453            0250 DSKINV =      $E453
0000            0260         *=    $600
                0270 ; *****************************
                0280 ; assume file begins at sector
                0290 ; $20 and extends to sector $60
                0300 ; *****************************
0600 A900       0310         LDA  #0
0602 8D0B03     0320         STA  DAUX2       ;hi sector number
0605 8D0803     0330         STA  DBYTLO      ;lo buffer length
0608 A980       0340         LDA  #$80        ;to load one sector
060A 8D0903     0350         STA  DBYTHI      ;  at a time
060D A950       0360         LDA  #$50        ;high byte of
060F 8D0503     0370         STA  DBUFHI      ;  buffer address
0612 A952       0380         LDA  #$52        ;get sector
0614 8D0203     0390         STA  DCOMND      ;command byte
0617 A920       0400         LDA  #$20        ;lo sector number
0619 8D0A03     0410         STA  DAUX1       ;  goes here
061C A900       0420 LOOP    LDA  #0          ;lo byte of buffer
061E 8D0403     0430         STA  DBUFLO      ;  address @ start
0621 2053E4     0440         JSR  DSKINV      ;read 1st sector
0624 A980       0450         LDA  #$80        ;move up 128 bytes
0626 8D0403     0460         STA  DBUFLO      ;  for buffer
0629 EE0A03     0470         INC  DAUX1       ;next sector
062C AD0A03     0480         LDA  DAUX1       ;are we done?
062F C960       0490         CMP  #$60
0631 B010       0500         BCS  FIN         ;yes
```

```
0633 2053E4 0510      JSR  DSKINV     ;no - read next sector
0636 EE0503 0520      INC  DBUFHI     ;raise buffer page
0639 EE0A03 0530      INC  DAUX1      ;next sector
063C AD0A03 0540      LDA  DAUX1      ;are we done?
063F C960   0550      CMP  #$60
0641 90D9   0560      BCC  LOOP       ;no
0643 60     0570 FIN  RTS             ;all finished
```

As we saw above, the further we get from the initial CIO routine, the more housekeeping we must take care of. In this program, we must handle the incrementing of the disk sectors and the buffer location after each read, and we must determine whether we are done by constantly comparing the sector number to the final sector desired, $60. This is what we meant when we compared the various I/O systems to the layers of an onion. The closer we get to the core, the more work we have to do, and the less the system handles for us.

At the very core is the Serial Input-Output system (SIO) itself. We accessed DSKINV in this program, but we could have called SIOV instead. However, before doing so, we would have had to set up the entire DCB instead of just the pertinent bytes, as we did. For instance, the serial bus ID would have had to be set to $31, in DDEVIC, and the timeout value to some reasonable value, like 45. Then we could have accomplished exactly the same results by replacing each call to DSKINV with a call to SIOV, but with the expense of still more housekeeping.

Note that both CIOV and DSKINV themselves call SIOV to actually accomplish the serial input and output, but they handle their respective housekeeping tasks before these calls. The further you get from CIO, the more precise your control of the system, but the more work for yourself. This is a general rule in computing — a high level language is the easiest to use, but gives you the least control of the system. As you gain more control, you also need to work harder. Well, you really didn't expect to get something for nothing, did you?

This concludes our discussion of disk I/O. You should now be completely familiar with how to get information to and from a disk drive, either using sequential or linked files. Experiment with these systems until you feel comfortable, since they are basic to many applications that you will want to try.

CHAPTER TEN
GRAPHICS AND SOUND FROM ASSEMBLY LANGUAGE

GRAPHICS

One of the most exciting and unique features of the ATARI computers is their excellent graphics. When compared with other popular microcomputers for quality of graphics, the ATARI is generally the clear winner. In fact, most arcade-type games available for several different computers look best on the ATARI, and advertising generally utilizes photographs taken from the screen generated on an ATARI.

"But," you say, "that's only available from BASIC." There is a common misconception among ATARI owners that the graphics commands are in the BASIC cartridge, and that commands like PLOT and DRAWTO can't be used without the BASIC cartridge in place. In fact, all of the graphics routines are located in the OS, and are therefore available from any language. We'll now see how to use these routines from assembly language.

Any program which requires such commands as GRAPHICS n, PLOT, or the other graphic commands, generally utilizes these many times throughout the program. It is therefore easiest to present these routines as a set of assembly language subroutines, which can be called from any program. These routines can be saved on a disk as a group and ENTERed into any program requiring graphics routines. To utilize the routines in the program, you'll generally have to load the X and Y registers and the accumulator with parameters that you'd like to implement, and then JSR to the appropriate routine. Note that this parameter passing is discussed in the comments to each routine, to make its use clear. Detailed discussion of the subroutines appears in the section following the program listings.

THE ASSEMBLY LANGUAGE GRAPHICS SUBROUTINES

```
            0100 ; ******************************
            0110 ; CIO equates
            0120 ; ******************************
0340        0130 ICHID  =     $0340
0341        0140 ICDNO  =     $0341
0342        0150 ICCOM  =     $0342
0343        0160 ICSTA  =     $0343
0344        0170 ICBAL  =     $0344
0345        0180 ICBAH  =     $0345
0346        0190 ICPTL  =     $0346
0347        0200 ICPTH  =     $0347
0348        0210 ICBLL  =     $0348
0349        0220 ICBLH  =     $0349
034A        0230 ICAX1  =     $034A
034B        0240 ICAX2  =     $034B
E456        0250 CIOV   =     $E456
            0260 ; ******************************
            0270 ; Other equates needed
            0280 ; ******************************
02C4        0290 COLOR0 =     $02C4
0055        0300 COLCRS =     $55
0054        0310 ROWCRS =     $54
02FB        0320 ATACHR =     $02FB
00CC        0330 STORE1 =     $CC
00CD        0340 STOCOL =     $CD
0000        0350        *=    $600
            0360 ; ******************************
            0370 ; The SETCOLOR routine
            0380 ; ******************************
```

```
             ; Before calling this routine,
             ; the registers should be set
             ; just like the BASIC SETCOLOR:
             ; SETCOLOR color,hue,luminance
             ;    stored respectively in
             ;   X reg.,accumulator,Y reg.
             SETCOL
0600 0A             ASL  A          ;need to multiply
0601 0A             ASL  A          ;  hue by 16, and
0602 0A             ASL  A          ;  add it to lum.
0603 0A             ASL  A          ;now hue is *16
0604 85CC           STA  STORE1     ;temporarily
0606 98             TYA             ;so we can add
0607 18             CLC             ;before adding
0608 65CC           ADC  STORE1     ;now have sum
060A 9DC402         STA  COLOR0,X   ;actual SETCOLOR
060D 60             RTS             ;all done
             ; ******************************
             ; The COLOR command
             ; ******************************
             ; For these routines, we will
             ; simply store the current COLOR
             ; in STOCOL, so the COLOR
             ; command simply requires that
             ; the accumulator hold the value
             ; "n" in the command COLOR n
             COLOR
060E 85CD           STA  STOCOL     ;that's it!
0610 60             RTS             ;all done
             ; ******************************
             ; The GRAPHICS command
             ; ******************************
             ; The "n" parameter of
             ; a GRAPHICS n command will be
             ; passed to this routine in the
             ; accumulator
             GRAFIC
0611 48             PHA             ;store on stack
0612 A260           LDX  #$60       ;IOCB6 for screen
0614 A90C           LDA  #$C        ;CLOSE command
0616 9D4203         STA  ICCOM,X    ; in command byte
0619 2056E4         JSR  CIOV       ;do the CLOSE
```

```
061C A260               LDX  #$60       ;the screen again
061E A903               LDA  #3         ;OPEN command
0620 9D4203             STA  ICCOM,X    ; in command byte
0623 A9AD               LDA  #NAME&255  ;name is "S:"
0625 9D4403             STA  ICBAL,X    ;  low byte
0628 A906               LDA  #NAME/256  ;  high byte
062A 9D4503             STA  ICBAH,X
062D 68                 PLA             ;get GRAPHICS n
062E 9D4B03             STA  ICAX2,X    ;graphics mode
0631 29F0               AND  #$F0       ;get high 4 bits
0633 4910               EOR  #$10       ;flip high bit
0635 090C               ORA  #$C        ;read or write
0637 9D4A03             STA  ICAX1,X    ;n+16,n+32 etc.
063A 2056E4             JSR  CIOV       ;setup GRAPHICS n
063D 60                 RTS             ;all done
               ; ******************************
               ; The POSITION command
               ; ******************************
               ; Identical to the BASIC
               ; POSITION X,Y command
               ; since X may be greater than
               ; 255 in GRAPHICS 8, we need to
               ; use the accumulator for the
               ; high byte of X
               POSITN
063E 8655               STX  COLCRS     ;low byte of X
0640 8556               STA  COLCRS+1   ;high byte of X
0642 8454               STY  ROWCRS     ;Y position
0644 60                 RTS             ;all done
               ; ******************************
               ; The PLOT command
               ; ******************************
               ; We'll use the X,Y, and A just
               ; like in the POSITION command
               PLOT
0645 203E06             JSR  POSITN     ;to store info.
0648 A260               LDX  #$60       ;for the screen
064A A90B               LDA  #$B        ;put record
064C 9D4203             STA  ICCOM,X    ;command byte
064F A900               LDA  #0         ;special case of
0651 9D4803             STA  ICBLL,X    ; I/O using the
```

```
0654 9D4903 1220        STA  ICBLH,X    ; accumulator
0657 A5CD   1230        LDA  STOCOL     ;get COLOR to use
0659 2056E4 1240        JSR  CIOV       ;plot the point
065C 60     1250        RTS             ;all done
            1260 ; ******************************
            1270 ; The DRAWTO command
            1280 ; ******************************
            1290 ; We'll use the X,Y, and A just
            1300 ; like in the POSITION command
            1310 DRAWTO
065D 203E06 1320        JSR  POSITN     ;to store info
0660 A5CD   1330        LDA  STOCOL     ;get COLOR
0662 8DFB02 1340        STA  ATACHR     ;keep CIO happy
0665 A260   1350        LDX  #$60       ;the screen again
0667 A911   1360        LDA  #$11       ;for DRAWTO
0669 9D4203 1370        STA  ICCOM,X    ;command byte
066C A90C   1380        LDA  #$C        ;as in XIO
066E 9D4A03 1390        STA  ICAX1,X    ; auxiliary 1
0671 A900   1400        LDA  #0         ;clear
0673 9D4B03 1410        STA  ICAX2,X    ; auxiliary 2
0676 2056E4 1420        JSR  CIOV       ;draw the line
0679 60     1430        RTS             ;all done
            1440 ; ******************************
            1450 ; The FILL command
            1460 ; ******************************
            1470 ; We'll use the X,Y, and A just
            1480 ; like in the POSITION command-
            1490 ; this is similar to DRAWTO
            1500 FILL
067A 203E06 1510        JSR  POSITN     ;to store info
067D A5CD   1520        LDA  STOCOL     ;get COLOR
067F 8DFB02 1530        STA  ATACHR     ;keep CIO happy
0682 A260   1540        LDX  #$60       ;the screen again
0684 A912   1550        LDA  #$12       ;for FILL
0686 9D4203 1560        STA  ICCOM,X    ;command byte
0689 A90C   1570        LDA  #$C        ;as in XIO
068B 9D4A03 1580        STA  ICAX1,X    ; auxiliary 1
068E A900   1590        LDA  #0         ;clear
0690 9D4B03 1600        STA  ICAX2,X    ; auxiliary 2
0693 2056E4 1610        JSR  CIOV       ;FILL the area
0696 60     1620        RTS             ;all done
```

```
               1630 ; ******************************
               1640 ; The LOCATE command
               1650 ; ******************************
               1660 ; We'll use the X,Y, and A just
               1670 ; like in the POSITION command
               1680 ; and the accumulator will
               1690 ; contain the LOCATEd color
               1700 LOCATE
0697 203E06    1710        JSR  POSITN    ;to store info.
069A A260      1720        LDX  #$60      ;the screen again
069C A907      1730        LDA  #7        ;get record
069E 9D4203    1740        STA  ICCOM,X   ;command byte
06A1 A900      1750        LDA  #0        ;special case of
06A3 9D4803    1760        STA  ICBLL,X   ; data transfer
06A6 9D4903    1770        STA  ICBLH,X   ; in accumulator
06A9 2056E4    1780        JSR  CIOV      ;do the LOCATE
06AC 60        1790        RTS            ;all done
               1800 ; ******************************
               1810 ; The screen's name
               1820 ; ******************************
06AD 53        1830 NAME   .BYTE "S:",$9B
06AE 3A
06AF 9B
```

DISCUSSION OF THE GRAPHICS SUBROUTINES

The first point to note about these routines is that they simply use the standard CIO equates, which we have seen so often before, plus six new ones. We don't need a whole new set of equates, since we're using the standard ATARI CIO routines for all of the graphics commands. Of the six new equates, two are simply storage locations: STOCOL is used to store the COLOR information used in several of the routines, and STORE1 is used for temporary storage of information. These are arbitrarily located at $CD and $CC respectively, but you may feel free to locate them at any safe memory location you choose. One such place would be $100 and $101, which are the bottom two locations of the stack. Another of the new equates is COLOR0, which is the first of the 5 locations used to store color information in the ATARI computers, found in decimal locations 708 to 712. The second is COLCRS, a 2-byte storage location at $55 and $56, which always stores the current column location of the cursor. Since in GRAPHICS 8 there are 320 possible horizontal locations, and we know that each single byte can store only 256 possible values, we need 2 bytes to store all possible horizontal positions of the cursor. However, in all graphics modes other than GRAPHICS 8, it is obvious that location $56 will always be equal to zero. The third new equate is ROWCRS, location $54, which simply keeps track of the vertical position of the cursor. No graphics mode has more than 192 possible vertical positions of the cursor, so only 1 byte is required to store this information. The final new equate is ATACHR, location $2FB, which is used to store the color of the line being drawn in both the FILL and DRAWTO routines.

These routines have been assembled using an origin of $600 for convenience. If you plan to use these in a larger assembly language program, just renumber these subroutines to some high line numbers, such as 25000 and up, and merge these routines with your program before assembling it. This way, you'll have all of the normal graphics commands available from assembly language, without needing to laboriously enter them into each program you write.

The first routine is the assembly language equivalent of the BASIC command SETCOLOR. We know that this is the standard form of the command in BASIC:

SETCOLOR color register, hue, luminance

In the assembly language subroutine, we first need to load the 6502 registers with the equivalent information. A typical calling routine to use this subroutine to simulate the BASIC command

```
SETCOLOR 2,4,10
```

would be as follows:

```
25  LDX #2
30  LDA #4
35  LDY #10
40  JSR SETCOL
```

We use the 6-letter form of the name SETCOL for SETCOLOR so that this routine will be compatible with all of the available assemblers for the ATARI. If the assembler you are using allows label names longer than 6 characters, feel free to use the whole routine name. This same convention will be used for all of the graphics routines — for instance, POSITN for POSITION.

To perform the SETCOLOR command, we need to add the luminance to 16 times the hue and store the result into the appropriate color register. To multiply the hue by 16, we'll simply use the accumulator and perform four ASL A instructions. Since each doubles the value contained in the accumulator, the result is 16 times the initial value. After the multiplication, we'll store the result into our temporary storage location and get the luminance into the accumulator with a TYA instruction, setting up for the addition. We then clear the carry bit, as usual prior to addition, and add the result of our previous multiplication to the luminance. Finally, we use the value in the X register, which is the color register desired, as an index into the five color register locations described above. Since we want to SETCOLOR 2 in this example, we loaded the X register with 2 before the call to the subroutine, and the color information is stored in $2C6.

The next routine, the COLOR command, is by far the easiest of all the routines. To call the COLOR equivalent of the BASIC command

```
COLOR 3
```

we simply need the following assembly language code:

```
25  LDA #3
30  JSR COLOR
```

The routine simply stores the color selected into our storage location for color, STOCOL, where it will be available for the other graphics routines which require it.

The GRAPHICS command is implemented similiarly. To mimic the BASIC command

```
GRAPHICS 23
```

we simply use the following assembly language code:

```
25 LDA #23
30 JSR GRAFIC
```

The first thing we need to do is store the graphics mode required. We could store it in STORE1, but pushing it onto the stack is quicker in this case; we don't need to do addition or multiplication, as we did in SETCOLOR. The next four lines of code simply close the screen as a device. This is for insurance. If the screen is already closed, we haven't hurt anything. However, if it's open and we try to reopen it, we'll get an error, so we close it first for insurance. Note that simply by using IOCB6 (loading the X register with $60), we specify the screen, using the default device number assigned by ATARI.

The remainder of the GRAPHICS command simply opens the screen in the particular graphics mode we desire. We again use IOCB6, storing the OPEN command in the command byte of the IOCB in line 830. The name of the screen is S:, and we load the address of this name into ICBAL and ICBAH. The graphics mode is then retrieved from the stack and stored in the second auxiliary byte. The only important bits of the graphics mode in ICAX2 are the lower 4 bits, which specify the graphics mode itself; in this case, GRAPHICS 7. The upper 4 bits control the clearing of the screen, the presence of the text window, and so on, as described in Chapter 8. In this case, we have added 16 to the graphics mode, to eliminate the text window. To isolate these bits, we AND the graphics mode with $F0, which yields the high nibble of the graphics mode. The OS requires that the high bit of this information be inverted, so next we EOR this nibble with $10, to flip the high bit. Finally, we set the low nibble of this byte to $C, to allow either reading or writing to the screen, and we store the byte in ICAX1 of the IOCB. The call to CIO completes our graphics routine and sets up the screen as we had wanted.

As we have already discussed, the POSITION command for GRAPHICS 8 requires 320 possible X locations, so we need 2 bytes to hold this large a number. Therefore, to simulate the command

```
POSITION 285,73
```

we will store the low byte of the X coordinate in the X register, the high byte in the accumulator, and the Y coordinate in the Y register, as follows:

```
25  LDX #30
30  LDA #1
35  LDY #73
40  JSR POSITN
```

Obviously, in any graphics mode other than GRAPHICS 8, the accumulator is always loaded with a zero prior to calling POSITN, and the X register simply contains the X coordinate. The routine itself simply stores the appropriate information into the required locations. The X coordinate is stored into COLCRS and COLCRS + 1, and the Y coordinate is stored into ROWCRS.

The PLOT command of

```
PLOT 258,13
```

in BASIC is simulated by the following code in assembly language:

```
25  LDX #3
30  LDA #1
35  LDY #13
40  JSR PLOT
```

This uses the same convention as the POSITN command above. In fact, the PLOT routine begins with a JSR POSITN, which stores the information passed to the routine into the correct locations for use by the OS following the call to CIO. Since we want to output to the screen, we use IOCB6, and the command byte is $B for put record. In this case, we simply want to output a single byte of information, so we use the special case of accumulator I/O accessed by setting the length of the output buffer to zero. Then we load the accumulator with the color information we want to plot, and the call to CIO plots the point for us.

The routines to DRAWTO and FILL are so similar that they will be discussed together. The calling sequence is identical to the PLOT and POSITION commands, so to mimic the BASIC command

```
DRAWTO 42,80
```

we use the sequence

```
25  LDX #42
30  LDA #0
35  LDY #80
40  JSR DRAWTO
```

To use the FILL command, simply change line 40 to JSR FILL.

The routine begins with a call to the POSITN routine to store the required information. The color information is then stored in ATACHR, and we use IOCB6 again, loading ICCOM with $11 for DRAWTO and with $12 for FILL. ICAX1 needs a $C, and we clear ICAX2 before completing the routine by calling CIO. These routines are absolutely analogous to the respective BASIC XIO commands which accomplish the same ends. For instance, to draw a line, we can use this command:

```
XIO 17,#6,12,0,"S:"
```

In this command, the 17 is the $11 command byte, the #6 is the IOCB number, the 12 is stored in ICAX1, the zero in ICAX2, and the device name is S:. Again, exactly the same XIO command can be use to FILL an area, by simply changing the 17 to 18 ($12).

The final routine, the LOCATE command, is virtually identical to the PLOT command, except that we use the get record command, rather than the put record command. The same use is made of the special single-byte accumulator I/O mode, by setting both ICBLL and ICBLH to zero. The calling routine to duplicate the BASIC command

```
LOCATE 10,12,A
```

is as follows:

```
25 LDX #10
30 LDA #0
35 LDY #12
40 JSR LOCATE
```

In this case, the accumulator will contain the color value found at the coordinates 10,12 following the call to the LOCATE routine, so a STA command could save this information, or it could be used immediately, by comparing it to some desired value, or in other ways.

This concludes the discussion of the assembly language counterparts to the BASIC graphics commands. Use them in some simple programs, and you'll see how soon they become familiar and how easy they are. In fact, they're almost as easy to use as the BASIC commands. However, since both BASIC and assembly language use the same OS routines to accomplish such operations as DRAWTO and FILL, don't expect that the assembly language routines will be much faster than the BASIC routines you are used to. They will be slightly faster, since you don't have to pay the overhead that BASIC requires in terms of time, but you will experience nowhere near the difference in speed that you have now come to expect when converting from BASIC to assembly language programming. To accomplish this kind of speedup, you'll have to write your own DRAWTO and FILL routines, using a totally different logic from that used by the ATARI OS. Such routines have been written and are much faster than the OS routines, but they are not in the public domain, and you'll have to write your own if speed is critical.

Now that you are becoming proficient in assembly language, you may want to change the ATARI central routines for your own purposes. If you want to try this, purchasing the OS listings and Technical User's Notes from ATARI is highly recommended. You can then look at the commented source code for the OS routines, and modify them for your own routines. Just include them as part of your own programs, making the modifications you would like. However, remember that the code for the OS belongs to ATARI. You can use such modifications in programs for your own use, but *be sure to get permission* from ATARI before trying to offer for sale any programs containing parts of ATARI's OS. One easy change to try is to allow plotting and drawing without checking for cursor out of range, which slows things down quite a bit. Just be sure that your program calculates the values correctly, or else....

Remember that anything possible from BASIC is also possible from assembly language. One frequently used example of this is animation by means of rotation of the color registers, possible using either the special GTIA modes, or the regular graphics modes. A very simple routine can rotate the standard color registers virtually instantaneously:

```
15  LDA $708
20  STA STOCOL
25  LDA $709
30  STA $708
35  LDA $710
40  STA $709
45  LDA $711
50  STA $710
55  LDA $712
60  STA $711
65  LDA STOCOL
70  STA $712
```

Now that you can draw detailed graphics from assembly language programs, this trick can be used to animate pictures with virtually no slowdown in program execution. For instance, implementation of a down-the-trench type game is simple, by letting rotation of the colors give the illusion of motion down the trench.

PLAYER-MISSILE GRAPHICS FROM ASSEMBLY LANGUAGE

Another exciting feature of the ATARI computers is player-missile graphics. We've already seen an example using an assembly language subroutine to move a player. But in that program, the entire setup for player-missile graphics was in BASIC, and only the routine to move the player was in assembly language. To show how to perform these same operations in a purely assembly language program, this BASIC program has been totally translated into assembly language and is presented below. In this program, decimal addresses are used for the most part, since that is the way the BASIC program was written; this assembly language program is as similiar to that BASIC program as is feasible.

```
                0100 ; *****************************
                0110 ; CIO equates
                0120 ; *****************************
0340            0130 ICHID  =     $0340
0341            0140 ICDNO  =     $0341
0342            0150 ICCOM  =     $0342
0343            0160 ICSTA  =     $0343
0344            0170 ICBAL  =     $0344
0345            0180 ICBAH  =     $0345
0346            0190 ICPTL  =     $0346
0347            0200 ICPTH  =     $0347
0348            0210 ICBLL  =     $0348
0349            0220 ICBLH  =     $0349
034A            0230 ICAX1  =     $034A
034B            0240 ICAX2  =     $034B
E456            0250 CIOV   =     $E456
                0260 ; *****************************
                0270 ; Other equates needed
                0280 ; *****************************
00CC            0290 YLOC   =     $CC       ;indirect addr. for Y
00CE            0300 XLOC   =     $CE       ;to remember X position
00D0            0310 INITX  =     $D0       ;initial X value
00D1            0320 INITY  =     $D1       ;initial Y value
0100            0330 STOTOP =     $100      ;temporary storage
D300            0340 STICK  =     $D300     ;hardware STICK(0) location
D000            0350 HPOSP0 =     $D000     ;horizontal pos. P0
0000            0360        *=    $600
                0370 ; *****************************
                0380 ; First, lower top of RAM
                0390 ; *****************************
0600 A56A       0400        LDA   106       ;get top of RAM
0602 8D0001     0410        STA   STOTOP    ;temporary storage
0605 38         0420        SEC             ;setup for subtract
0606 E908       0430        SBC   #8        ;save 8 pages for PMG
```

```
0608 856A   0440        STA  106        ;tell ATARI-new RAMTOP
060A 8D07D4 0450        STA  54279      ;PMBASE
060D 85CF   0460        STA  XLOC+1     ;to erase PM RAM
060F A900   0470        LDA  #0         ;  put indirect
0611 85CE   0480        STA  XLOC       ;  addr. here
            0490 ; ******************************
            0500 ; Next, reset GRAPHICS 0
            0510 ; ******************************
0613 A900   0520        LDA  #0         ;GRAPHICS 0
0615 48     0530        PHA             ;store on stack
0616 A260   0540        LDX  #$60       ;IOCB6 for screen
0618 A90C   0550        LDA  #$C        ;CLOSE command
061A 9D4203 0560        STA  ICCOM,X    ; in command byte
061D 2056E4 0570        JSR  CIOV       ;do the CLOSE
0620 A260   0580        LDX  #$60       ;the screen again
0622 A903   0590        LDA  #3         ;OPEN command
0624 9D4203 0600        STA  ICCOM,X    ; in command byte
0627 A9ED   0610        LDA  #NAME&255  ;name is "S:"
0629 9D4403 0620        STA  ICBAL,X    ;  low byte
062C A906   0630        LDA  #NAME/256  ;  high byte
062E 9D4503 0640        STA  ICBAH,X
0631 68     0650        PLA             ;get GRAPHICS 0
0632 9D4B03 0660        STA  ICAX2,X    ;graphics mode
0635 29F0   0670        AND  #$F0       ;get high 4 bits
0637 4910   0680        EOR  #$10       ;flip high bit
0639 090C   0690        ORA  #$C        ;read or write
063B 9D4A03 0700        STA  ICAX1,X    ;n+16,n+32 etc.
063E 2056E4 0710        JSR  CIOV       ;set up GRAPHICS 0
            0720 ; ******************************
            0730 ; Now set up PMG
            0740 ; ******************************
0641 A978   0750        LDA  #120       ;initial X value
0643 85D0   0760        STA  INITX      ;  put in place
0645 A932   0770        LDA  #50        ;initial Y value
0647 85D1   0780        STA  INITY      ;  put in place
0649 A92E   0790        LDA  #46        ;double line
064B 8D2F02 0800        STA  559        ;  resolution
            0810 ; ******************************
            0820 ; Now clear out PM area of RAM
            0830 ; ******************************
064E A000   0840        LDY  #0         ;use as counter
0650 A900   0850        LDA  #0         ;byte to be stored
```

```
               0860 CLEAR
0652 91CE      0870        STA  (XLOC),Y  ;clear 1st byte
0654 88        0880        DEY            ;is page finished?
0655 D0FB      0890        BNE  CLEAR     ;page not done yet
0657 E6CF      0900        INC  XLOC+1    ;page is done
0659 A5CF      0910        LDA  XLOC+1    ;on to next page
065B CD0001    0920        CMP  STOTOP    ;are we done?
065E F0F2      0930        BEQ  CLEAR     ;one more page
0660 90F0      0940        BCC  CLEAR     ;keep going
               0950 ; ****************************
               0960 ; Now we'll insert player into
               0970 ; the appropriate place in the
               0980 ; PMG RAM area
               0990 ; ****************************
0662 A56A      1000        LDA  106       ;1st, calculate
0664 18        1010        CLC            ;  correct Y posit.
0665 6902      1020        ADC  #2        ;PMBASE+512(2 pages)
0667 85CD      1030        STA  YLOC+1    ;high byte of YLOC
0669 A5D1      1040        LDA  INITY     ;add Y screen coordinate
066B 85CC      1050        STA  YLOC      ;for low byte
066D A000      1060        LDY  #0        ;as a counter
               1070 INSERT
066F B9F006    1080        LDA  PLAYER,Y  ;get byte of player
0672 91CC      1090        STA  (YLOC),Y  ;put it in place
0674 C8        1100        INY            ;for next byte
0675 C008      1110        CPY  #8        ;are we done?
0677 D0F6      1120        BNE  INSERT    ;no
0679 A5D0      1130        LDA  INITX     ;get initial X
067B 8D00D0    1140        STA  53248     ;tell ATARI
067E 85CE      1150        STA  XLOC      ;also here
0680 A944      1160        LDA  #68       ;make player red
0682 8DC002    1170        STA  704       ;  as in BASIC
0685 A903      1180        LDA  #3        ;to enable player-
0687 8D1DD0    1190        STA  53277     ;  missle graphics
               1200 ; ****************************
               1210 ; The main loop-very short
               1220 ; ****************************
               1230 MAIN
068A 209A06    1240        JSR  RDSTK     ;read stick-move player
068D A205      1250        LDX  #5        ;to control the
068F A000      1260        LDY  #0        ;  player, we
               1270 DELAY
```

```
0691 88     1280          DEY              ; have to add
0692 D0FD   1290          BNE  DELAY       ; a delay - this
0694 CA     1300          DEX              ; routine slows
0695 D0FA   1310          BNE  DELAY       ; things down
0697 4C8A06 1320          JMP  MAIN        ;and do it again
            1330 ; ******************************
            1340 ;Now read the joystick #1
            1350 ; ******************************
            1360 RDSTK
069A AD00D3 1370          LDA  STICK       ;get joystick value
069D 2901   1380          AND  #1          ;is bit 0 = 1?
069F F016   1390          BEQ  UP          ;no - 11,12 or 1 o'clock
06A1 AD00D3 1400          LDA  STICK       ;get it again
06A4 2902   1410          AND  #2          ;is bit 1 = 1?
06A6 F020   1420          BEQ  DOWN        ;no - 5,6 or 7 o'clock
06A8 AD00D3 1430 SIDE     LDA  STICK       ;get it again
06AB 2904   1440          AND  #4          ;is bit 3 = 1?
06AD F02E   1450          BEQ  LEFT        ;no - 8,9 or 10 o'clock
06AF AD00D3 1460          LDA  STICK       ;get it again
06B2 2908   1470          AND  #8          ;is bit 4 = 1?
06B4 F02F   1480          BEQ  RIGHT       ;no - 2,3 or 4 o'clock
06B6 60     1490          RTS              ;joystick straight up
            1500 ; ******************************
            1510 ;Now move player appropriately
            1520 ;starting with upward movement
            1530 ; ******************************
06B7 A001   1540 UP       LDY  #1          ;setup for moving byte 1
06B9 C6CC   1550          DEC  YLOC        ;now 1 less than YLOC
06BB B1CC   1560 UP1      LDA  (YLOC),Y    ;get 1st byte
06BD 88     1570          DEY              ;to move it up one position
06BE 91CC   1580          STA  (YLOC),Y    ;move it
06C0 C8     1590          INY              ;now original value
06C1 C8     1600          INY              ;now set for next byte
06C2 C00A   1610          CPY  #10         ;are we done?
06C4 90F5   1620          BCC  UP1         ;no
06C6 B0E0   1630          BCS  SIDE        ;forced branch!!!
            1640 ; ******************************
            1650 ;Now move player down
            1660 ; ******************************
06C8 A007   1670 DOWN     LDY  #7          ;move top byte first
06CA B1CC   1680 DOWN1    LDA  (YLOC),Y    ;get top byte
06CC C8     1690          INY              ;to move it down screen
```

```
06CD 91CC              STA  (YLOC),Y  ;move it
06CF 88                DEY            ;now back to starting value
06D0 88                DEY            ;set for next lower byte
06D1 10F7              BPL  DOWN1     ;if Y>=0 keep going
06D3 C8                INY            ;set to zero
06D4 A900              LDA  #0        ;to clear top byte
06D6 91CC              STA  (YLOC),Y  ;clear it
06D8 E6CC              INC  YLOC      ;now is 1 higher
06DA 18                CLC            ;setup for forced branch
06DB 90CB              BCC  SIDE      ;forced branch again
                 ; ******************************
                 ;Now side-to-side - left first
                 ; ******************************
06DD C6CE        LEFT  DEC  XLOC      ;to move it left
06DF A5CE              LDA  XLOC      ;get it
06E1 8D00D0            STA  HPOSP0    ;move it
06E4 60                RTS            ;back to MAIN - we're done
                 ; ******************************
                 ;Now right movement
                 ; ******************************
06E5 E6CE        RIGHT INC  XLOC      ;to move it right
06E7 A5CE              LDA  XLOC      ;get it
06E9 8D00D0            STA  HPOSP0    ;move it
06EC 60                RTS            ;back to MAIN - we're done
                 ; *****************************
                 ; DATA statements
                 ; *****************************
06ED 53          NAME   .BYTE "S:",$9B
06EE 3A
06EF 9B
06F0 FF          PLAYER .BYTE 255,129,129,129,129,129,129,255
06F1 81
06F2 81
06F3 81
06F4 81
06F5 81
06F6 81
06F7 FF
```

This program uses many of the routines we have already discussed — an erasing routine to clear out the player-missile area of memory, reading the joystick and moving the player, and the

GRAPHICS 0 command. Here, we simply put them all together into one large program which performs all of the tasks necessary to implement a simple example of player-missile graphics in assembly language.

Since it is analogous to the BASIC program we have already written, the assembly version begins by lowering RAMTOP by 8 pages to make room for player-missile memory. Lines 400 to 440 perform this function; line 410 stores the old value of RAMTOP for the erasing routine later. Line 450 tells the ATARI the location of PMBASE, the new value of RAMTOP. We'll use XLOC and XLOC + 1 as a temporary indirect address location on page zero, to help in the erase routine. Lines 520 to 710 then reset a GRAPHICS 0 screen below the new location of RAMTOP. Lines 750 to 780 simply store the initial values of X and Y, the screen coordinates where we want the player to appear. These values will be used later, and these lines are here only to keep the analogy with the BASIC program. There is no need to store these values; we could just as easily have used the numbers directly later in the program. However, either way works, and it is slightly easier to change the program later if it is written in this manner. Lines 790 and 800 set up double-line resolution, and then we erase the entire player-missile area in lines 840 to 940.

In the BASIC program, the place in memory where we insert the player to achieve the correct Y positioning on the screen is:

```
PMBASE+512+INITY
```

We know that 512 bytes above PMBASE is 2 pages, since each page contains 256 bytes. Therefore, we know that the high byte of this address, in our assembly language program, must be 2 higher than PMBASE. In lines 1000 to 1030, we get PMBASE, add 2, and store the result in YLOC + 1, the high byte of the Y position in memory. The low byte is simply INITY, the initial Y position. Remember, the farther down the screen you want the player to appear, the higher in memory the player must be stored.

To insert the player into the correct place in memory, we read one byte at a time from the table of data called PLAYER, and store it using indirect addressing into the memory location we just set up. When Y, our counter, equals 8, we're done, since we started at zero and have only 8 bytes to transfer. If our player had been larger, we simply would have changed the single byte in line 1110 to 1 higher than the number of bytes in the player. The initial X position of the player is read from INITX and stored in the horizontal position register for player zero, 53248. It is also stored in XLOC for use by the move-player routine.

We then make the player red by storing the number 68 into the color register for player zero, 704, and enable player-missile graphics by storing a 3 into 53277, GRACTL.

The main loop of this program is simplicity itself. We JSR to the routine which reads the joystick and moves the player, and then we enter a short delay loop. If we leave out this loop, the player moves so quickly that we can't control it at all! Next, we simply loop back to read the joystick and move the player again.

Obviously, if you want to add some interest to this program, you can insert your own program logic into this main loop, to detect player-playfield collisions, or create obstacles, or anything else you may want in your game. If you are going to lengthen the program, however, you should change the origin to somewhere higher in memory. As it is, this program already occupies virtually all of page 6, so if you make it any larger without changing the origin, it will begin to overwrite DOS and you'll not be able to save or load it. Just change the origin to $6000 or some other safe high memory location. To test the program after assembling it, just type BUG to enter the debugger, and then type G600 for the original version (or G6000 if you change the origin).

Now that you've seen how to implement player-missile graphics from assembly language, you should be able to write your own programs utilizing these same techniques. By doing this, as we've already seen from the need to insert a delay loop in the above program, you'll speed things up enormously and create smooth motion of players to greatly enhance your games. Have fun!

CREATING SOUND ON ATARI COMPUTERS

We will begin our discussion of sound by learning how the ATARI produces sounds, and then we'll write an assembly language subroutine to mimic the BASIC SOUND command.

Let's first look at the equates used for sound generation. The POKEY chip is responsible for the creation of all sounds in the ATARI computers, and it resides in memory from $D200 to $D2FF. The sounds which add so much to enjoyment of games, and can even add to the ease of use of business programs if used properly, are divided into four voices. Each voice is controlled by two registers, located in pairs from $D200 to $D207. The first of each pair is the frequency control, and the second of each pair controls both the volume and distortion of the sound produced by that channel. These are:

```
100 AUDF1  = $D200 ;audio channel 1 frequency
110 AUDC1  = $D201 ;audio channel 1 control
120 AUDF2  = $D202 ;audio channel 2 frequency
130 AUDC2  = $D203 ;audio channel 2 control
140 AUDF3  = $D204 ;audio channel 3 frequency
150 AUDC3  = $D205 ;audio channel 3 control
160 AUDF4  = $D206 ;audio channel 4 frequency
170 AUDC4  = $D207 ;audio channel 4 control
```

The respective frequency registers control the pitch of the sound or note being played. These registers actually divide the sound frequency by the number stored here. That is, if we store a 12 here, then the frequency produced is one-twelfth of the input frequency.

The input frequency is controlled by the initialization of POKEY, by setting AUDCTL. The following chart describes the use of each bit in AUDCTL:

Bit	Use
0	Set to switch main clock from 64 kHz to 15 kHz
1	Set to insert high-pass filter into channel 2
2	Set to insert high-pass filter into channel 1
3	Set to join channels 4 and 3 for 16-bit resolution
4	Set to join channels 2 and 1 for 16-bit resolution
5	Set to clock channel 3 with 1.79 MHz
6	Set to clock channel 1 with 1.79 MHz
7	Set to convert the 17 bit poly-counter into 9 bits

What does all this mean? Let's take it one bit at a time. Suppose you store a 10 into the frequency register of voice 1. We already know that this will cause one pulse to come out of that voice for each ten going in. Bit 0 of AUDCTL can switch the frequency of the incoming pulses between 64 kHz and 15 kHz. Kilohertz (kHz) stands for thousands of cycles, or pulses, per second. Obviously, if AUDCTL is set with bit 0 equal to zero, then the output frequency of voice 1 is 6.4 kHz. However, if we store a one into AUDCTL bit 0, then the output frequency of voice 1 will be 1.5 kHz, and a markedly lower tone will result. Bits 5 and 6 work exactly the same way, but if we set these to 1, the voices controlled by them, channels 3 and 1 respectively, produce much higher pitches, since they would be clocked at 1.79 MHz (millions of cycles per second), many times faster than either of the above frequencies.

The two bits 1 and 2 of AUDCTL insert high-pass filters into channels 2 and 1, respectively. These high-pass filters are clocked by channels 4 and 3, respectively. That is, only sounds with a higher frequency than those currently playing in channel 3 will be heard in channel 1, and only sounds with a higher frequency than those currently sounding in channel 4 will be heard in channel 2. Some spectacular special effects are possible using these high-pass filters, and you'll certainly want to experiment to see what can be done.

Since the frequency is stored in a single byte, the ATARI voices are limited to about a 5-octave range. However, using AUDCTL, it is possible to pair together sets of two voices using bits 3 or 4. This allows the two frequency registers for these voices to form a 16-bit number, giving a nine-octave range. This decreases the number of voices available, but it would be perfectly feasible to produce one 9-octave voice and two 5-octave voices, or even two 9-octave voices. When two frequency registers are combined into one, the higher of the two frequency registers controls the high byte of the 2-byte number and the lower of the two controls the lower byte.

The high bit of AUDCTL controls the polynomial counter. This is perhaps the most difficult concept of sound generation on the ATARI computers to grasp. Basically, the **poly-counters** produce a random sequence of pulses which repeats after some time. The higher the number of bits in the poly-counter, the longer the random sequence will be before the pattern repeats. There are three different poly-counters in the ATARI, and they all function as follows.

Suppose you want some noise to be produced. Music is regular in tone, but noise is irregular and is harder to produce. The ATARI generates noise by producing a random sequence of pulses from the poly-counter and effectively ANDing these pulses together with the output of the frequency registers discussed above. Only when both pulses are ON is a sound produced. For example, if the frequency register says a pulse, or sound, should be produced, but the random pattern from the poly-counter is off at that time, no sound is produced. Therefore, although the output of the frequency register's divide-by-n system is a pure frequency, ANDing these pulses with the random sequence generated by the poly-counters produces noise. It should be apparent that different poly-counters produce different noise sounds, and the poly-counter used can be selected by bit 7 of AUDCTL, which is a 9-bit or 17-bit poly-counter.

Furthermore, the distortion, or noise type, of the sound produced can also be changed by the distortion setting, much as in BASIC. The distortion is controlled by the upper 3 bits of the control register for each voice, AUDC1–4. These bits essentially choose how the sound is to be treated and which of the three poly-counters is to be used for the distortion, as follows:

Bits 765	Meaning
0 0 0	Select using 5-bit, then 17-bit poly, divide by 2
0 E 1	Select using 5-bit poly, divide by 2
0 1 0	Select using 5-bit, then 4-bit poly, divide by 2
1 0 0	Select using 17-bit poly, divide by 2
1 E 1	No poly-counters used, just divide by 2
1 1 0	Select using 4-bit poly, divide by 2

E in the above table means that bit can be either a 1 or a 0. First, of course, the clock rate is divided by the frequency. Let's look at an example of how the distortion system works. We'll assume that the clock is running at 15 kHz, that we've stored 30 into the appropriate frequency register, and that the 3 high bits of the control register for that voice are 010. First, the clock rate is divided by the frequency register, in this case, 15000/30 = 500 Hz. Next, since the distortion bits are 010, the pulses at 500 Hz output from this operation are effectively ANDed with the output of the 5-bit polynomial counter. The output of this operation is then effectively ANDed with the output of the 4-bit poly-counter, and the frequency of the pulses successfully getting through this entire operation is divided by 2 to produce the final distorted sound.

It should be obvious that with so many options to choose from, the ATARI is capable of many, many, many different sound effects. Some experimentation is clearly in order here. You may hear some really far-out sounds being produced!

A SOUND SUBROUTINE

Creation of sounds on the ATARI computers is extremely easy in BASIC, since the SOUND command allows us to turn any of the four available voices on or off at any desired distortion, pitch, volume, and frequency. Exactly the same functions are available from assembly language. We can write a subroutine to mimic the effects of the BASIC SOUND command, much as we did for the graphics commands in the previous section.

```
            0100 ; ******************************
            0110 ; SOUND equates
            0120 ; ******************************
D200        0130 AUDF1  =    $D200     ;audio 1 freq.
D201        0140 AUDC1  =    $D201     ;audio 1 control
D208        0150 AUDCTL =    $D208     ;audio control
D20F        0160 SKCTL  =    $D20F     ;serial port control
0101        0170 STORE2 =    $101      ;temporary store
0000        0180        *=   $600
            0190 ; ******************************
            0200 ; The SOUND command
            0210 ; ******************************
            0220 ; Prior to calling this routine,
            0230 ; the X register should contain
            0240 ; the voice desired, the accum.
            0250 ; should contain the distortion,
            0260 ; the Y register should
            0270 ; contain the volume desired,
            0280 ; and STORE2 should contain the
            0290 ; desired frequency.
            0300 SOUND
```

```
0600 48     0310     PHA            ;store distortion
0601 8A     0320     TXA            ;double voice value
0602 0A     0330     ASL  A         ;  for offset to
0603 AA     0340     TAX            ;  voice control
0604 AD0101 0350     LDA  STORE2    ;frequency into
0607 9D00D2 0360     STA  AUDF1,X   ;  right channel
060A 8C0101 0370     STY  STORE2    ;for use later
060D A900   0380     LDA  #0        ;to initialize the
060F 8D08D2 0390     STA  AUDCTL    ;  POKEY chip
0612 A903   0400     LDA  #3        ;  set these as
0614 8D0FD2 0410     STA  SKCTL     ;  indicated
0617 68     0420     PLA            ;retrieve distortion
0618 0A     0430     ASL  A         ;now multiply by
0619 0A     0440     ASL  A         ;  16 to get the
061A 0A     0450     ASL  A         ;  distortion into
061B 0A     0460     ASL  A         ;  the high nibble
061C 18     0470     CLC            ;setup for add
061D 6D0101 0480     ADC  STORE2    ;add the volume
0620 9D01D2 0490     STA  AUDC1,X   ;into right voice
0623 60     0500     RTS            ;that's all
```

In this program, we double the value originally stored in the X register, which will select the particular voice to be used. This is because the sound registers are arranged in pairs, so each of the control registers and each of the frequency registers are two bytes apart in memory. The frequency is then retrieved from STORE2, which is used because we need four pieces of information for sound, and between the X and Y registers and the accumulator, we have only three storage locations at hand. The frequency is then stored in the appropriate frequency register in line 360, and the volume is temporarily stored until we need it. We then initialize POKEY in lines 380 to 410 and convert the distortion to the upper bits of the accumulator, adding the volume to obtain the number which needs to be stored in the appropriate audio control register to produce the sound desired. That's all there is to it!

However, sound generation in assembly language suffers from the same problem it has in BASIC: no duration can be specified for the sound produced. Either the SOUND command in BASIC, or our equivalent routine in assembly language, simply starts the sound. We must turn the sound off after some predetermined time has elapsed, to create the note or sound effect we want. To turn off the sound, just store a zero into the appropriate control register, AUDC1–4. To measure the time elapsed, use either the timers already discussed, at locations 18 to 20 decimal, or use a routine like the one we used in the player-missile example for short delays:

```
     LDX #50
     LDY #0
LOOP DEY
     BNE LOOP
     DEX
     BNE LOOP
```

The time delay in this kind of routine is controlled by the initial value loaded into the X register; the larger the number, the longer the delay. These nested loops would take forever to execute if we programmed the counterpart to this routine in BASIC, but after assembly the delay is very short. Try it to see how fast 256 X 50, or 12,800 loops, can be.

COUNTDOWN TIMERS

The third way of keeping track of time on the ATARI computers involves the use of countdown timers. AUDF1, AUDF2, and AUDF4 can act as countdown timers in the following way. When any nonzero value is stored into STIMER ($D209), the values stored in AUDF1, AUDF2, and AUDF4 begin to decrease. Each of these three registers has an appropriate vector location, as outlined below:

Timer	Vector location
AUDF1	$210, $211 (VTIMR1)
AUDF2	$212, $213 (VTIMR2)
AUDF4	$214, $215 (VTIMR4)

If we place the address of a short routine to turn off sound into one of these vector locations, an interrupt will be generated when the appropriate timer has counted down to zero, and control will shift to your routine to turn off the sound. Just remember to end this routine with an RTI rather than an RTS. Before using this type of timer, you must place the appropriate value into the interrupt request enable byte, IRQEN ($D20E). To enable VTIMR1, set bit 0 of IRQEN; to enable VTIMR2, set bit 1 of IRQEN; and to enable VTIMR4, set bit 2 of IRQEN.

You should now be able to create any sounds available from BASIC using assembly language, and here's one which can't be produced from BASIC because of the speed required. If bit 4 of any of the audio control registers is set to 1, a short "pop" can be heard. This is caused by pushing the speaker cone out once, compressing the air in front of the speaker, which you hear as a "pop." If we set and reset this bit quickly, we can produce a sound just by compressing air in front of the speaker. Try this:

```
LOOP LDA #16
     STA AUDC1
     (insert delay for frequency)
     LDA #31
     STA AUDC1
     (insert delay for frequency)
     JMP LOOP
```

The speaker on your TV or monitor will vibrate back and forth, producing sound in a totally different way from that described above. In this example, we are simply moving the speaker directly in order to produce sound.

PART FOUR
APPENDIXES

APPENDIX ONE
THE 6502 INSTRUCTION SET

One caution must be mentioned with regard to the use of the 6502 instruction set. Every computer language has its own **syntax**, the way each command must be written for the computer to understand it. Different assemblers for the ATARI require slightly different syntax, described in detail in Chapter 6.

In this Appendix, we shall learn the commands which the 6502 can execute. The complete set of commands for the 6502 is generally referred to as the **6502 instruction set**. Here the instructions are described in alphabetical order for easy reference. Along with each instruction you will find:

1. Examples of its use
2. Descriptions of the various addressing modes available for that instruction (see Chapter 5)
3. The effect of each instruction upon the various flags in the processor status register (see Chapter 3)

ADC Add with Carry

THE INSTRUCTION

As discussed in Chapter 4, each instruction for the 6502 is abbreviated into a three-letter code called a **mnemonic**. The first instruction, Add with Carry, is the only instruction available for carrying out addition of two numbers. Here's an example of its use:

```
.       ;these represent
.       ;some previous
.       ;instructions
ADC #2  ;add 2 to the contents of the accumulator
.
.
```

Let's examine what happens when we execute this instruction. The 6502 takes whatever is already contained in the accumulator and adds to it the decimal number 2. The resulting sum remains in the accumulator, awaiting further instructions. However, there is a complication. Remember, the name of this instruction is Add with Carry. What about the carry? Remember that the Carry bit is one of the flags in the processor status register. Whenever the ADC instruction is encountered, the Carry bit is added to the sum in the accumulator. Let's suppose that just before encountering this instruction, we had stored a zero in the accumulator, and the Carry bit was a zero, as well. The sum stored in the accumulator following the execution of this instruction would still be 2, as described. However, if the Carry bit had been a 1, then the sum would have been 3. Schematically, this addition is as follows:

Accumulator	Carry Bit	Instruction	=>	Sum	Carry Bit
0	0	ADC #2	=>	2	0
0	1	ADC #2	=>	3	0

Note that if the Carry bit is initially 1, it is reset to zero after the ADC instruction is executed. This makes sense, since we've already used the carry. If it wasn't reset to zero by the 6502, we might use the Carry bit twice without realizing it.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The 6502 also sets the Carry bit appropriately, as in the following examples:

Accumulator	Carry Bit	Instruction	=>	Sum	Carry Bit
253	0	ADC #6	=>	3	1
253	1	ADC #6	=>	4	1

In the first example, 253 + 6 = 259, but we know that the largest number the accumulator can hold is 255. The number 256 represents zero, with a carry of one, so 259 is 3 with a carry of 1. The number 3 is stored in the accumulator and the Carry bit is set.

In the second example, the Carry bit starts at 1, so when it is added into the sum, the sum is 4. Remember, after the Carry bit is used, the 6502 resets it to zero. So why does example 2 end up with the Carry bit set? The sum was greater than 255, so the Carry bit was set before execution of the instruction was completed.

We have already discussed the ability of the 6502 to operate in decimal mode. Note here that the largest number the accumulator can store in decimal mode is 99. The Carry bit is set following an ADC instruction in decimal mode if the sum exceeds 99.

Several other flags in the processor status register are also conditioned by the ADC instruction. The Overflow bit, V, is set when bit 7, the most significant bit, is changed because of the addition. The Negative flag, N, is set if the addition produces a number greater than 127; that is, when the most significant bit, bit 7, is a 1. Remember, to the 6502, any number between 128 and 255 is a negative number, because its most significant bit is a 1. Finally, the Zero flag, Z, is set if the result of the addition is zero. Some examples of these conditions are shown below:

A	N	Z	C	V	Instruction	=>	A	N	Z	C	V	Comments
2	0	0	0	0	ADC #3	=>	5	0	0	0	0	Straight addition
2	0	0	1	0	ADC #3	=>	6	0	0	0	0	Remember: add the carry!
2	0	0	0	0	ADC #254	=>	0	0	1	1	0	C and Z flags set
2	0	0	0	0	ADC #253	=>	255	1	0	0	0	N and V flags set
253	1	0	0	0	ADC #6	=>	3	0	0	1	1	C and V set; N reset
125	0	0	1	0	ADC #2	=>	128	1	0	0	1	N and V set; C reset

It should now be apparent that by testing the various flags, we can easily obtain a great deal of information about the results of an addition. Instructions for testing these flags are provided and are used frequently in most assembly language programs.

ADDRESSING MODES

The ADC instruction allows any of the same eight addressing modes discussed in Chapter 5 for the LDA instruction. They are illustrated below, with the number of cycles and bytes used for each instruction:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	ADC #2	2	2	A + #2
Absolute	ADC $3420	4	3	A + contents of memory $3420
Zero Page	ADC $F6	3	2	A + contents of memory $F6
Zero Page,X	ADC $F6,X	4	2	A + contents of memory $F6 + X
Absolute,X	ADC $3420,X	4	3	A + contents of memory $3420 + X
Absolute,Y	ADC $3420,Y	4	3	A + contents of memory $3420 + Y
Ind. Indir.	ADC ($F6,X)	6	2	A + contents of addr. at $F6 + X
Indir. Ind.	ADC ($F6),Y	5	2	A + contents of (address at $F6) + offset Y

For more complete explanations of these addressing modes, please see Chapter 5.

AND The Logical AND Instruction

THE INSTRUCTION

The AND instruction performs a logical AND of the accumulator and the operand. A logical AND takes two numbers and compares them bit for bit. For each bit, if both numbers being compared contain a 1, the result contains a 1 as well; if either number contains a zero, the result contains a zero in that bit. Let's look at an example in which the accumulator contains the number 5:

```
AND #$0E
```

The easiest way to visualize the AND operation is to convert each of the two numbers being compared to binary nomenclature, as follows:

Hex.		Binary		
#5	=	#%00000101		
#$0E	=	#%00001110		
result	=	#%00000100	=	#4

We can easily see that the only bits set to 1 in the result are those in which both numbers being compared have a 1, in this example, bit 2. Let's try another example. The accumulator contains 147, and this is the instruction:

```
AND #$1D
```

As above, we convert to binary:

#147	=	#%10010011		
#$1D	=	#%00011101		
result	=	#%00010001	=	#17

This fairly straightforward instruction is used frequently, so you should now try an exercise by yourself. We'll assume the accumulator contains the number #$BC, and this is the instruction:

```
AND #234
```

See if you can work out the answer for yourself.

The AND instruction is used to mask a byte in a specific way. For instance, suppose you want to know the value of the low nibble of a number. To find out, you simply AND the number with #$0F. Since the high nibble of this number is zero and the low nibble is all ones, the result will be equal to the low nibble of the number you started with. Similarly, to obtain the high nibble you only have to AND with #$F0. Now let's present the answer to the above problem:

```
 #$BC  =  #%10111100
 #234  =  #%11101010
result =  #$10101000  =  #168
```

EFFECTS ON THE PROCESSOR STATUS REGISTER

The AND instruction sets the Zero flag if the result is equal to zero, or resets the Zero flag if the result is not equal to zero. This instruction also sets the Negative flag if the result is greater than 127, or resets this flag if the result is less than 128. Several examples of this follow:

A	Z	N	Instruction	=>	A	Z	N	Meaning
#5	0	0	AND #8	=>	0	1	0	Z set by result=0
#$FE	0	1	AND #$5F	=>	#$5E	0	0	N reset by result<127

ADDRESSING MODES

The AND instruction can also be written in any of the same eight addressing modes discussed in Chapter 5 for the LDA instruction.

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	AND #2	2	2	A&#2
Absolute	AND $3420	4	3	A&contents of memory $3420
Zero Page	AND $F6	3	2	A&contents of memory $F6

Mode	Instruction	Cycles	Bytes	Meaning
Zero Page,X	AND $F6,X	4	2	A&contents of memory $F6 + X
Absolute,X	AND $3420,X	4	3	A&contents of memory $3420 + X
Absolute,Y	AND $3420,Y	4	3	A&contents of memory $3420 + Y
Ind. Indir.	AND ($F6,X)	6	2	A&contents of addr. at $F6 + X
Indir. Ind.	AND ($F6),Y	5	2	A&contents of (address at $F6) + offset Y

ASL Arithmetic Shift Left

THE INSTRUCTION

The ASL instruction utilizes the carry bit in the processor status register as a ninth bit for a number, and pushes each bit in a number one place to the left; thus, the name **arithmetic shift left**. A zero is pushed into the least significant bit, and the most significant bit of the original number ends up in the Carry bit. A picture is worth a thousand words here:

C		76543210		
0	<=	10110101	<=	0
1		01101010		

Let's look at another example:

C		76543210		
0	<=	01101101	<=	0
0		11011010		

"What is the purpose of this instruction?" you may ask. Well, let's look more closely at the last example. The original number,

#%01101101, is #109 in decimal form; following the ASL the resulting number is #%11011010, which is #218 in decimal form. We have doubled the number with a single instruction! Since each position in a binary number is exactly twice the value of the position to its immediate right, a shift to the left doubles the value of each bit. This is an extremely easy way of multiplying numbers by powers of 2; just ASL once for each power of 2 required.

> **CAUTION:** Although it seems fairly easy to multiply a number by 2 using the ASL instruction, you must be extremely careful and correct for overflow into the Carry bit. An example of this is the first set of numbers above. We started with #%10110101, which is the decimal number 181, and following our ASL we had #%01101010, which is the decimal number 106. Now we know that 106 is not 2 times 181. In fact, 2 times 181 is 362. Note that this is exactly 106, the result we got, plus 256. Remember, in that example, we concluded with a carry of 1, which represents 256 when using the ASL instruction for multiplication. You must take this carry into account whenever you use the ASL instruction.

EFFECTS ON THE PROCESSOR STATUS REGISTER

As we have already seen, the Carry bit will contain the most significant bit of the original number. The Negative flag will be set equal to the most significant bit of the result (bit 6 of the original number). The Zero flag will be set if the result is equal to zero, and will be reset for any other result. In the examples shown below, we'll use the accumulator addressing mode.

A	N	C	Z	Instruction	=>	A	N	C	Z
#128	1	0	0	ASL A	=>	#0	0	1	1
#64	0	0	0	ASL A	=>	#128	1	0	0
#192	1	0	0	ASL A	=>	#128	1	1	0
#8	0	0	0	ASL A	=>	#16	0	0	0

ADDRESSING MODES

Examples using the five addressing modes for the ASL instruction are listed below:

Mode	Instruction	Cycles	Bytes	Meaning
Accumulator	ASL A	2	1	ASL value in accumulator
Absolute	ASL $3420	6	3	ASL contents: memory $3420
Zero Page	ASL $F6	5	2	ASL contents: memory $F6
Zero Page,X	ASL $F6,X	6	2	ASL contents: memory $F6 + X
Absolute,X	ASL $3420,X	7	3	ASL contents: memory $3420 + X

Note that this instruction takes quite a while to execute. In fact, the Absolute,X addressing mode of the ASL instruction requires 7 machine cycles to execute, the longest of any of the instructions in the entire 6502 set. However, we need to think about speed in absolute terms. Using this instruction allows us to perform a multiplication by 2 in 3.92 microseconds, even in this slowest of modes. Still pretty fast, especially when compared with BASIC!

BCC Branch on Carry Clear

THE INSTRUCTION

In BASIC, we can distinguish between two different types of commands which both transfer the control of a program to a new line number. The first is the straight GOTO command; we know that the number of the next line to be executed will always follow the command GOTO:

```
45 GOTO 90
50 .
60 .
90 ? "This line follows 45."
```

After line 45 is executed, line 90 is the next line under all conditions. This is called an **unconditional** transfer of control.

A second type of transfer in BASIC utilizes the comparison and branching abilities of the computer:

```
45 IF X<32 THEN 90
50 .
   .
   .
90 ? "This line sometimes follows 45."
```

In this example, if the value of the variable X is less than 32 in line 45, then a branch in the flow of the program is taken, and line 90 is executed next. If X is equal to or greater than 32, then the branch is not taken, and line 50 is executed next. Thus, the program flow depends on certain conditions established within the program, and for this reason, this type of transfer is called **conditional** transfer of control. In this example, the **condition** is the value of the variable X.

The BCC instruction means that if the Carry bit is clear (equal to zero), then program control must branch to a specific location. As in BASIC, if the condition is not met — in this example, if the carry bit is equal to 1 — the line next executed is not the line specified in the instruction, but rather the next line in the program.

In virtually all cases, branches in assembly language are specified by labels. A **label** can be almost any name you want to give to a specific line in an assembly language program. Let's look at a short example:

```
     BCC SKIP
     LDA $0345
     .
     .        ;other lines
     .        ;of the program
     .
SKIP LDA $4582
```

We'll take this one line at a time. First, we see the instruction BCC SKIP. The Carry bit can either be a one or a zero at the time this instruction is executed. Let's first assume that the Carry bit is set to 1. When this line is executed, the BCC test fails; that is, since the Carry is not clear, the branch to SKIP is not taken. The next line executed loads the accumulator from address $0345. The program then proceeds with the next line and then the next, and so on.

Now let's assume that when the BCC is executed, the Carry bit is zero. In this case, the branch is taken, since the Carry is clear, and the next line executed loads the accumulator from memory location $4582. Then the line immediately following the SKIP line will be executed.

It should be obvious by now that in addition to the actual lines of instructions in an assembly language program, the values of each of the flags in the processor status register are an important factor in understanding a program. Instructions provided in the instruction set allow us to control these bits directly.

Still another similarity between the BCC instruction and a conditional branch in BASIC is that the branch can either be forward or backward in the program. The example above is a forward branch. To see what a backward branch looks like, let's look at the following example:

```
SKIP LDA $0254
     .
     .
     BCC SKIP
```

In this case, if the Carry bit is equal to zero, we branch backward from the BCC to SKIP. If it is equal to 1, the line following the BCC will be executed next.

There is an important limitation to such branches in assembly language. The BCC instruction can transfer control no further than 127 bytes forward or backward in the program. Only 1 byte is used to hold the value of the branch and any 1-byte number greater than 127 is recognized as a negative number, so if we try to branch ahead 130 bytes, the 6502 recognizes this as a negative branch of 255 − 130 = 125 bytes. Instead of jumping ahead in our program, we'd be back some considerable distance over ground we'd already traveled. Most assemblers will detect the error if we try to branch too far, and report it as such at the time of assembly, but this may be very time-consuming. It's a lot easier to try to avoid this problem while writing our programs.

One caution about relative branches: most assemblers will allow a branch of the form

```
BCC +7
```

which tells the program counter to branch forward 7 bytes if the Carry is clear when this line is executed.

CAUTION: THIS IS VERY BAD PROGRAMMING PRACTICE!!!

The problem comes when you try to read your program, or, heaven forbid, try to change it. If you insert a line shortly after this, the branch taken by this BCC will probably be wrong. The use of labels for branch target points makes your program much easier for you to read and lessens the chance of errors. Don't, under penalty of long, long, long hours of debugging, write branches like the above!

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The only addressing mode for the BCC instruction is the Relative mode, and we have already seen how it works. Branches are either taken or not, depending on the value of the Carry bit. Branches are either forward or backward relative to the current position of the program counter, which normally points to the start of the line immediately following the BCC instruction. The BCC instruction requires 2 bytes and takes 2 machine cycles to execute.

BCS Branch on Carry Set

THE INSTRUCTION

BCS is the exact opposite of the BCC instruction. The branch is taken when the Carry bit is equal to 1 and is not taken when the Carry bit is equal to zero. In all other respects, BCS and BCC are identical. Refer to the BCC instruction for further details.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The only addressing mode available for the BCS instruction is the Relative mode. Its use was described above for the BCC instruction. The BCS instruction uses 2 bytes and takes 2 machine cycles to execute.

BEQ Branch on Equal to Zero

THE INSTRUCTION

The BEQ instruction is similar to the BCC and BCS instructions, but differs in one important way. Instead of using the Carry bit, the BEQ instruction uses the Zero bit as the determining factor in evaluating whether or not to take a branch. If the Zero bit is equal to 1, the branch is taken. Remember, the Zero bit will be equal to 1 only if some operation resulted in an answer of zero; thus the name Branch on Equal to Zero. If the Zero bit is equal to zero, the previous result was not equal to zero, and the branch would not be taken. This may be a little confusing at first. The best way to remember it is to understand that the Zero bit is a flag, and the flag is set when a certain condition is met. In the case of the Zero flag, the condition is a result of zero, which sets the Zero flag. Remember, we're testing for the flag being set, not for the flag being equal to zero.

```
     LDA #0
     BEQ SKIP
     .
     .
SKIP LDA $2F
```

In this example, when we load the accumulator with zero, the Zero bit is set in the processor status register. Therefore, the line executed after the BCC is SKIP, not the next line in the program. If instead we load the accumulator with 1, the branch will not be taken, and the program flow will be in order.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The only available mode for the BEQ instruction is the Relative mode, discussed in detail in the section on the BCC instruction. The BEQ instruction needs 2 bytes of storage and requires 2 machine cycles to execute.

BIT Test Bits in Memory with Accumulator

THE INSTRUCTION

The BIT instruction performs an AND between the number stored in the accumulator and the number stored in another memory location addressed in the instruction, but it is different from the usual AND instruction in one very important way. The AND instruction performs the AND operation between the number in the accumulator and a number in a memory location and stores the result in the accumulator. The BIT instruction performs the AND, but does not store the result in the accumulator. "So what good is it?" you may ask.

Remember that the AND instruction does two things. First, it performs the AND and stores the result in the accumulator. Second, it sets and resets various flags in the processor status register. The BIT instruction performs the first function without storing the number, but it also performs the second, discussed below.

EFFECTS ON THE PROCESSOR STATUS REGISTER

Three flags in the processor status register are conditioned by the BIT instruction. The Negative flag is set to the value of bit 7, the most significant bit, of the byte stored in the memory location being tested, and the V (oVerflow) flag is set equal to the value of bit 6 of the same byte. The Zero flag is set if the result of the AND operation is equal to zero; it is reset if the result is not equal to zero. Some examples of the BIT instruction are given below, along with their effects on the processor status register flags. For the purpose of these examples, let's assume that memory location $0345 contains the value #$F3.

A	N	V	Z	Instruction	=>	A	N	V	Z
#128	1	0	0	BIT $0345	=>	#128	1	1	0
#5	0	0	0	BIT $0345	=>	#1	1	1	0
#4	0	0	0	BIT $0345	=>	#0	1	1	1
#3	0	0	1	BIT $0345	=>	#3	1	1	0

This instruction is used primarily when you want to learn something about a value stored somewhere in memory without disturbing the value stored in the accumulator. For instance, you can easily learn whether the number in memory is negative, because after the BIT operation, the N flag will be set if the number was negative. Similarly, you can learn whether bit 6 of the number is a one or a zero by looking at the V flag after the BIT operation. Finally, you can determine whether an AND between the accumulator and the number in memory results in a zero value by testing the Z flag. Note that an AND operation between any number and zero will produce a zero result; so any time the number addressed in memory equals zero, the result of the BIT operation will be equal to zero and the Z flag will be set. This is quite a lot of information for an instruction which at first glance appeared to do nothing!

ADDRESSING MODES

Only two addressing modes are available for the BIT instruction, Absolute and Zero Page.

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	BIT $3420	4	3	A&contents of memory $3420
Zero Page	BIT $F6	3	2	A&contents of memory $F6

Since the BIT instruction simply compares the values stored in the accumulator and in a specific memory location, these two modes are sufficient for any use of BIT you may require. Between them they address the entire memory space of the computer.

BMI Branch on Minus

THE INSTRUCTION

BMI is another of the conditional branch instructions in the 6502 instruction set. It utilizes the Negative flag in the processor status register; the branch is taken if the Negative flag is set and is not taken if this flag is equal to zero. In all other respects, BMI is similar to the BCC instruction already discussed. Please read that discussion for details of conditional branch instructions, and read the discussion below of the BPL instruction for a caution concerning these two instructions.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The only addressing mode for the BMI instruction is the Relative mode, discussed for the BCC instruction. The BMI instruction requires 2 bytes of memory and takes 2 machine cycles to execute.

BNE Branch on Not Equal to Zero

THE INSTRUCTION

BNE is the exact opposite of the BEQ instruction. With the BNE instruction, the branch is taken if the Zero flag is equal to zero; that is, when the previous result was not equal to zero. The branch is not taken when the Zero flag is equal to 1. Refer to the discussion of the BCC instruction for details.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

Only the Relative addressing mode, discussed under the BCC instruction, is available for the BNE instruction. The BNE instruction requires 2 bytes of memory and takes 2 machine cycles to execute.

BPL Branch on Plus

THE INSTRUCTION

This instruction is the exact opposite of BMI. The branch is taken following the BPL instruction if the Negative flag is equal to zero and is not taken if this flag is equal to 1. One caution should be mentioned for this pair of instructions: in order for the branch to be correctly determined, the Negative flag must, of course, have been correctly conditioned prior to executing either the BMI or BPL instruction. Since not all other instructions condition the Negative flag, be sure that you use one which does correctly condition this flag before utilizing either the BMI or BPL instruction.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

Only the Relative addressing mode is available for the BPL instruction. Please see the discussion of the BCC instruction for details. The BPL instruction requires 2 bytes of memory and takes 2 machine cycles to execute.

BRK Break

THE INSTRUCTION

The BRK instruction is somewhat analogous to the BASIC STOP command. We know that the STOP command causes the program being executed to stop; at that point control is returned to the BASIC cartridge, which signals that it is back in command by telling you what happened and printing READY to the screen.

The primary use of the BRK instruction is in debugging your program after it has been written, but before it's working quite the way you intended. In BASIC, we frequently go through this debugging process by inserting STOP instructions at various places in the program and then running the program to see if we get to the various STOPs. In assembly language programming, BRKs can be used in exactly the same way. You can insert a number of BRK instructions throughout your program and then try to run it. If you don't reach the BRK instructions, you know that your program is "hanging up" somewhere prior to that instruction. Most available debuggers will print out the values of the 6502 registers whenever a BRK instruction is encountered in a program; this makes the job of debugging somewhat easier. Even with such aids, debugging a long assembly language program should not be attempted by the faint of heart, at least not unless there is a big slice of time available with nothing else to do.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

Only one addressing mode is available for the BRK instruction, the Implied mode. This is a single-byte mode, and for the BRK instruction it requires 7 machine cycles to execute.

BVC Branch on Overflow Clear

THE INSTRUCTION

BVC is another of the conditional branch instructions in the 6502 instruction set; it utilizes the Overflow flag of the processor status register. If the V flag is set, the branch is not taken, but if the V flag is clear (equal to zero), the branch is taken. See the discussion of the BCC instruction for further details on conditional branch instructions.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The BVC instruction utilizes only the one addressing mode common to all of the conditional branch instruction, the Relative mode. The instruction requires 2 bytes and takes 2 machine cycles to execute.

BVS Branch on Overflow Set

THE INSTRUCTION

The BVS instruction is the exact opposite of the BVC instruction. If the Overflow flag in the processor status register is set (equal to 1), the branch is taken, and if the V flag is clear (equal to zero), the branch is not taken. You can find the details of relative branching in the discussion of the BCC instruction.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The BVS instruction utilizes only the Relative mode of addressing.

CLC Clear the Carry Bit

THE INSTRUCTION

The CLC instruction has a direct and constant effect on a flag in the processor status register. It clears the Carry bit — that is, sets it equal to zero, whether it was initially zero or 1. It's most commonly used prior to addition with the ADC instruction. Since the ADC instruction always adds the Carry bit into the sum, we generally need to be sure that this bit is equal to zero before addition. This is the only way to be sure that 2 plus 1 will equal 3, and not 4 at times. Almost universally, the typical set of instructions to add two numbers will be:

```
LDA #2
CLC
ADC #1
```

We load the accumulator with the first number — in this case, 2. Then we clear the Carry bit, because we have no way of knowing for certain the value of this bit at any time without a lot more code. By setting the Carry bit to zero, we know we'll get an accurate sum. We complete the operation by adding with carry the number 1. Remember that the sum, 3 in this case, will be stored in the accumulator at the completion of the addition; other lines will probably do something with this sum, such as store it in memory or utilize it in some further operation.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The only effect CLC has on the processor status register is to always set the Carry bit to zero.

ADDRESSING MODES

The sole addressing mode available for the CLC instruction is the Implied mode. The instruction is only 1 byte long and requires only 2 machine cycles to execute.

CLD Clear the Decimal Flag

THE INSTRUCTION

As we have already discussed, the 6502 can operate in either the decimal mode or the binary mode. The CLD instruction clears the decimal flag in the processor status register and resets the 6502 to operate in binary rather than decimal mode. In this book, we'll use the binary mode for most examples, but we'll briefly cover the decimal mode here. In this mode, each nibble of a byte represents a single decimal digit. The coding scheme is referred to as **binary-coded decimal** and is given below:

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

The difference between binary and decimal coding shows up when addition or subtraction takes place. For example, let's try the following code:

```
LDA $0345  ;contains #$59
CLC        ;put before an add instruction
ADC $0302  ;contains #$13
```

What number is contained in the accumulator after execution of these lines? We would normally think that this number should be #$6C, and if the addition were done in the binary mode we'd be correct. But suppose that we had first put the 6502 into the decimal mode. In that case, the number stored in the accumulator would be 72, because the 6502 would interpret the bytes at locations $0345 and \$0302 as binary-coded decimal numbers, and would add them in decimal mode: 59 + 13 = 72.

As was mentioned before, the examples in this book are all in binary form. The decimal mode may be used, however, when a result needs to be displayed to the screen quickly. Since each nibble of the number represents a decimal digit, by masking the appropriate digit using the AND instruction and then sending it to the screen, we can display a number quickly, without the need to interconvert between binary and decimal nomenclature.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The CLD instruction directly sets the Decimal flag of the processor status register to zero, thus clearing it. No other effects occur.

ADDRESSING MODES

As with all instructions operating directly on flags in the processor status register, the only addressing mode available for the CLD instruction is the Implied mode. The instruction requires only 1 byte of memory and takes 2 machine cycles to execute.

CLI Clear the Interrupt Flag

THE INSTRUCTION

The CLI instruction operates directly on the processor status register to clear the Interrupt flag; that is, to set the flag equal to zero. This presumably follows some instruction which had set this flag. Remember that when the Interrupt flag is set, maskable interrupts are disabled. This is important for ATARI programming because several different types of interrupts are used routinely in many programs, and if the I flag is set, they cannot occur. The vertical blank interrupt and the display list interrupt are included in this category. The CLI instruction allows these interrupts to occur.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The CLI instruction operates directly to clear the I flag and has no effect on any of the other flags in the processor status register.

ADDRESSING MODES

The CLI instruction utilizes only one addressing mode, the Implied mode, and takes 1 byte of memory and 2 machine cycles to execute.

CLV Clear the Overflow Flag

THE INSTRUCTION

CLV is just like the two previous instructions, CLD and CLI, which clear a flag in the processor status register. In the case of the CLV instruction, the Overflow flag is cleared.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The CLV instruction clears the Overflow flag, but has no effect on any of the other flags in the processor status register.

ADDRESSING MODES

Like the other instructions operating directly on the processor status register, CLV has only one mode of address, the Implied mode.

CMP Compare Memory and the Accumulator

THE INSTRUCTION

In BASIC, we can compare two values and determine whether they are equal, whether A is greater than B, or whether A is less than B. Generally, this is done in an IF statement of this type:

```
35 IF A<B THEN ...
```

Using the 6502 instruction set, we can also compare two numbers, although in a slightly different way. One of the numbers to be compared must be in the accumulator, and the other may be anywhere in the memory of the computer. The instruction CMP subtracts the contents of the specified memory location from the value stored in the accumulator and sets the various processor status register flags appropriately after this subtraction. Note that the value stored in the accumulator does **not** change following the CMP instruction. The subtraction only sets the flags; it does not change the value originally stored in the accumulator. Knowing the values of the flags in the processor status register, we can deduce the results of the comparison. Let's first discuss the changes in the processor status register, and then we'll work on some CMP examples.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The CMP instruction sets the Zero flag if the number in memory and the number stored in the accumulator are equal. If they are not equal, the Zero flag is reset to zero. The Negative flag is set following the CMP instruction if the result of the subtraction is greater than 127 (that is, the number has its most significant bit equal to one). Otherwise, the Negative flag is reset to zero. The Carry flag is set when the value stored in memory is less than or equal to the number stored in the accumulator. C is reset when the number stored in memory is less than that stored in the accumulator. Let's look at a few examples of the CMP instruction, assuming that memory location $0345 contains #26:

A	Z	N	C	Instruction	=>	A	Z	N	C	Comment
#26	0	0	0	CMP $0345	=>	#26	1	0	1	Z,C set by A = $0345
#48	0	0	0	CMP $0345	=>	#48	0	0	1	Z reset;C set by A > $0345
#130	0	1	0	CMP $0345	=>	#192	0	0	1	N reset by A-$0345 < 128
#8	0	0	0	CMP $0345	=>	#8	0	1	0	N set by A-$0345 > 127

In the first example, the two numbers being compared were equal, causing the subtraction to produce a result of zero. This result set the Zero flag. Since the number in memory was equal to the number in the accumulator, the Carry flag was also set. In the second example, the Zero flag was not set, since the result did not equal zero; the Negative flag was not set, since the result was positive (+22); and the Carry bit was set, since the number in memory was less than the number in the accumulator. In the third example, the number began as a negative, with the Negative flag set. Since the result was not equal to zero, the Zero bit was reset. The result of the subtraction, 130 minus 26, was the positive number 104, so the Negative flag was reset. Finally, since the value stored in $0345 was less than the value of #130 stored in the accumulator, the Carry bit was set.

This third example demonstrates an important point about the CMP instruction. A negative number is clearly less than a positive number, but the result of this comparison made it appear as if the negative number was larger. The CMP instruction **does not** use signed arithmetic; it simply compares two numbers between zero and 255, treating both numbers as positive integers. If you use signed arithmetic, you'll need to correct for this when comparing two numbers.

In the fourth example, the Zero bit was not set, since the result was not equal to zero. The Negative bit was set, since #8 − #26 = #238, which is a negative number. Finally, since the number in the accumulator was smaller than the number in memory, the Carry bit was reset with zero.

We can also look at these effects to determine how to test for the various results. The table below describes several values of the accumulator and memory, along with branch instructions which will be taken following a CMP instruction. The code looks like this:

```
LDA #...
CMP $....
B.. destination
```

Now we'll see which branches will be taken using various values for the two numbers.

A	Mem.	BCS, BPL, BNE
8	8	BEQ, BCS, BPL
9	8	BCS, BPL, BNE
8	9	BMI, BCC, BNE

Now we can see how to structure some code which will compare two numbers and take appropriate action, depending on the results obtained. Let's suppose that we want to duplicate the following BASIC code in assembly language:

```
25 IF R<B THEN GOTO Q
```

In ATARI BASIC, this means that the branch to line Q will be taken if the value of the variable R is less than the value of the variable B at the time line 25 is executed. In assembly language, the same code looks like this:

```
LDA R  ;load A from memory location R
CMP B  ;compare to memory location B
BCC Q  ;take branch if Carry reset
```

ADDRESSING MODES

CMP uses the same eight addressing modes discussed in Chapter 5 for the LDA instruction.

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	CMP #2	2	2	A-#2
Absolute	CMP $3420	4	3	A-contents of memory $3420
Zero Page	CMP $F6	3	2	A-contents of memory $F6
Zero Page,X	CMP $F6,X	4	2	A-contents of memory $F6 + X
Absolute,X	CMP $3420,X	4	3	A-contents of memory $3420 + X
Absolute,Y	CMP $3420,Y	4	3	A-contents of memory $3420 + Y

Mode	Instruction	Cycles	Bytes	Meaning
Ind. Indir.	CMP ($F6,X)	6	2	A-contents of addr. at $F6 + X
Indir. Ind.	CMP ($F6),Y	5	2	A-contents of (address at $F6) + offset Y

CPX Compare Index Register X with Memory

THE INSTRUCTION

This instruction compares the values in the X register and a specific memory location, in contrast to the CMP instruction, which compares the values in the accumulator and a memory location. In all other respects, the CPX and CMP instructions are identical. CPX is used primarily to test the value in register X, especially when it is being used as an index, in order to determine if it has reached the final desired value. For example, when the X register is being used as a loop counter and you wish to determine when to branch out of the loop, the CPX instruction, followed by an appropriate branch instruction, will give you the answer.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the two numbers being compared are equal, the Zero flag will be set (made equal to 1); otherwise, it will be reset (made equal to zero). If the result of the subtraction is greater than 127, the Negative flag will be set; otherwise, it will be reset. Finally, if the number in memory is less than or equal to the number stored in the X register, the Carry flag will be set; otherwise, it will be reset. Please refer to the CMP instruction for more details.

ADDRESSING MODES

The CPX instruction utilizes the three addressing modes outlined below.

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	CPX #2	2	2	X-#2
Absolute	CPX $3420	4	3	X-contents of memory $3420
Zero Page	CPX $F6	3	2	X-contents of memory $F6

CPY Compare Index Register Y with Memory

THE INSTRUCTION

CPY is identical to CPX or CMP in all respects, except that it uses the Y instead of the X register or the accumulator. As with the CPX instruction for the X register, when you need to determine whether the index register Y has reached a certain value, use the CPY instruction. Refer to the CMP and CPX sections of this Appendix for details.

EFFECTS ON THE PROCESSOR STATUS REGISTER

Please refer to the CMP instruction for a detailed description of the effects of CPY on the processor status register.

ADDRESSING MODES

The CPX and CPY instructions are identical in their addressing modes, examples of which are shown below:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	CPY #2	2	2	Y-#2
Absolute	CPY $3420	4	3	Y-contents of memory $3420
Zero Page	CPY $F6	3	2	Y-contents of memory $F6

Please refer to Chapter 5 for details of these addressing modes.

DEC Decrement Memory

THE INSTRUCTION

In order to decrement (decrease by 1) any memory location, the DEC instruction may be used. There are actually two ways to decrement a memory location. The first, and by far the easiest, is to use DEC directly. The second, and by far the more cumbersome, is to load the accumulator from that memory location, subtract 1, and then store the resulting number back into the original memory location. You can see why a DEC instruction was included in the 6502 instruction set.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the decrementing process results in a number equal to zero in the memory location addressed, the Zero flag in the processor status register will be set. If the result is any number but zero, the Zero flag will be reset. If the number resulting from the DEC instruction is negative (greater than 127), the Negative flag will be set; otherwise, it will be reset. It is therefore possible to determine when a decrementing instruction has produced either a zero or negative result without ever loading the number in question into the accumulator: simply check the status of either the Z or the N flag in the processor status register.

ADDRESSING MODES

Four addressing modes are available for the DEC instruction, as listed below:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	DEC $3420	6	3	DEC Contents of memory $3420
Zero Page	DEC $F6	5	2	DEC Contents of memory $F6
Zero Page,X	DEC $F6,X	6	2	DEC Contents of memory $F6 + X
Absolute,X	DEC $3420,X	7	3	DEC Contents of memory $3420 + X

DEX Decrement the X Register

THE INSTRUCTION

DEX specifically decrements the X register and is used primarily when the X register is being used as the index of a loop. Each time through the loop you decrement the register once. When the Z flag is set following this decrementing, you can branch out of the loop, knowing it has completed the predetermined number of cycles.

EFFECTS ON THE PROCESSOR STATUS REGISTER

Like the DEC instruction, the DEX instruction will set or reset both the Negative flag and the Zero flag in the processor status register. By testing these flags, a programmer can determine the state of the X register.

ADDRESSING MODES

Only one addressing mode is available for the DEX instruction, and as you might expect, it is the Implied mode, since the addressing can be inferred by the nature of the instruction. The instruction requires only 1 byte and takes 2 machine cycles to execute.

DEY Decrement the Y Register

THE INSTRUCTION

DEY is the Y-register counterpart to the DEX instruction. It decrements the Y register by 1.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The effects of the DEY instruction are the same as those of the DEC and DEX instructions already discussed.

ADDRESSING MODES

Like the DEX instruction, the DEY instruction uses only the Implied mode, requires 1 byte of memory, and takes 2 machine cycles to execute.

EOR Exclusive Or

THE INSTRUCTION

The EOR instruction is most like the AND instruction. Remember that AND performs a bit-by-bit AND: if a bit is set in both of the numbers being compared, the bit will be set in the resulting answer. The EOR instruction performs a bit-by-bit EOR. If a bit is set in one, and only one, of the numbers addressed, it is set in the answer. If the bit is set in both or neither, there is a zero in that position of the answer. The resulting number is stored in the accumulator. An example of this is shown below:

```
LDA #133
EOR #185
```

Again, the simplest way to determine the correct answer for the EOR operation is to visualize the numbers in their binary form:

Dec.		Binary		
#133	=	#%10000101		
#186	=	#%10111010		
Result	=	#%00111111	=	#63

Bit 7, which was set in both numbers, is equal to zero in the answer. Similarly, bit 6, which is equal to zero in both numbers, is also equal to zero in the answer. Only those bits which were set in only one of the numbers are set in the answer — bits 0 through 5.

The most common use of the EOR instruction is in complementing a number. To do this, we EOR the number with #$FF, a number in which each bit is set. For instance, to complement the number 143, we EOR it with #$FF:

```
  #143 = #%10001111
  #$FF = #%11111111
 ------------------
Result = #%01110000 = #112
```

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the resulting number, residing in the accumulator, is negative (greater than 127), the Negative flag is set; otherwise, it is reset. If the result of the EOR instruction is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

The EOR instruction utilizes the same 8 addressing modes as does the LDA instruction; these are detailed in Chapter 5. Brief examples are given below:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	EOR #2	2	2	A EOR #2
Absolute	EOR $3420	4	3	A EOR memory $3420
Zero Page	EOR $F6	3	2	A EOR memory $F6
Zero Page,X	EOR $F6,X	4	2	A EOR memory $F6 + X
Absolute,X	EOR $3420,X	4	3	A EOR memory $3420 + X
Absolute,Y	EOR $3420,Y	4	3	A EOR memory $3420 + Y
Ind. Indir.	EOR ($F6,X)	6	2	A EOR addr. at $F6 + X
Indir. Ind.	EOR ($F6),Y	5	2	A EOR (address at $F6) + offset Y

INC Increment Memory

THE INSTRUCTION

The INC instruction is the exact opposite of the DEC instruction, causing the value stored in any addressed memory location to be increased by 1.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The INC instruction sets the Negative flag if the resulting number is greater than 127; otherwise, it resets the Negative flag. It also sets the Zero flag if the resulting number is equal to zero; otherwise, it resets the Zero flag.

ADDRESSING MODES

INC utilizes the same four addressing modes utilized by the DEC instruction. Consult the examples below:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	INC $3420	6	3	INC contents of memory $3420
Zero Page	INC $F6	5	2	INC contents of memory $F6
Zero Page,X	INC $F6,X	6	2	INC contents of memory $F6 + X
Absolute,X	INC $3420,X	7	3	INC contents of memory $3420 + X

INX Increment the X Register

THE INSTRUCTION

The INX instruction is the exact opposite of the DEX instruction. It increments the value stored in the X register by 1 and is used to increase the value of the index register in loops, as shown in Chapters 7 to 10.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the increment causes the X register to be equal to zero, the Zero flag is set; otherwise, it is reset. If the resulting number is negative, the Negative flag is set; otherwise, it is reset.

ADDRESSING MODES

As with the other instructions of this type, the only addressing mode available is the Implied mode, requiring 1 byte of memory and 2 machine cycles to execute.

INY Increment the Y Register

THE INSTRUCTION

Similar to the INX instruction just discussed, and the exact opposite of the DEY instruction, INY causes the Y register to be increased by 1.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the results of the increment cause the number stored in the Y register to be negative, the Negative flag is set; otherwise, it is reset. If the Y register equals zero following the increment, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

The only addressing mode available for the INY instruction is the Implied mode, taking 1 byte of memory and 2 machine cycles to execute.

JMP Jump to Address

THE INSTRUCTION

We have discussed several examples of conditional transfer of control using branching instructions. These are the counterparts of the BASIC IF statement. However, we know that BASIC also has an unconditional transfer of control instruction, the GOTO statement:

```
30 GOTO Q
40 .
```

We know that the line executed after line 30 will be line Q, not line 40. This does not depend on anything within the program; that is, it is totally unconditional. Every time line 30 is executed, line Q will be the next line executed.

Assembly language has its counterpart of the GOTO statement — the JMP instruction. This is its form:

```
JMP Q
```

It works exactly like the BASIC example above. Every time the JMP is executed, line Q will be the next line executed, regardless of any conditions established while running the program. This is a totally unconditional transfer of control.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The JMP instruction has only two addressing modes. The first is the Absolute mode, in which the jump takes place to a specific memory location, as described in the example above. This mode uses 3 bytes of memory and requires 3 maching cycles to execute.

The second addressing mode is the Indirect mode, which is used only for the JMP instruction. To use this mode, we must first set up an indirect memory location somewhere in memory. Let's suppose that we would like to JMP indirectly to memory location $0620. We must first decide where in memory we will store the indirect address; here we'll use locations $0423 and $0424. We first store the byte #$20 in memory location $0423 and the byte #$06 in memory location $0424. Remember, for the 6502, the low byte comes first, followed by the high byte of the address. The indirect jump is then of this form:

```
JMP ($0423)
```

The parentheses indicate that it is an indirect jump; and the low address of the indirect address is given in the instruction. It is also possible to set an address label for $0423, in which case the instruction could be written like this:

```
JMP (Q)
```

Figure A-1 illustrates the indirect jump graphically. The indirect JMP instruction requires 3 bytes and takes 5 machine cycles to execute.

Location	Contents	
0421	A9	
0422	B3	
0423	2Ø	= $Ø62Ø
0424	Ø6	
0425	95	
0426	DE	

Fig. A-1 Indirect jump JMP ($Ø423)

JSR Jump to Subroutine

THE INSTRUCTION

In BASIC, we can use subroutines to perform repetitive tasks, and we can do the same thing in assembly language. The BASIC command GOSUB has an exact counterpart in assembly language — JSR. This instruction pushes the value of the program counter onto the stack, where it remains until the subroutine is completed. The value is then pulled off the stack so that the program may continue execution in the appropriate place. In the program below, first the LDA is executed, then the subroutine at Q, and then the STA.

```
LDA #124
JSR Q
STA $4657
```

One note about the use of subroutines in assembly language is appropriate here. In BASIC, programs run more rapidly if frequently used subroutines are all located as close to the beginning of the program as possible. Because the BASIC interpreter starts scanning the program for the target line from the beginning, this practice avoids the need to search the whole program to find the subroutine. In assembly language, this is not the case. At the time of assembly, the actual address of the subroutine is placed following the code for the JSR instruction, so the subroutine can be located anywhere. In practice, it's a good idea to group all your subroutines together, usually at the end of your assembly language program, both for readability and for ease of access for changes, but it's up to you. The program will execute in exactly the same way no matter where you locate the subroutines.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The JSR instruction utilizes only the Absolute addressing mode. It uses 3 bytes of memory and 6 machine cycles to execute.

LDA Load the Accumulator

THE INSTRUCTION

The LDA instruction loads the accumulator with a number, either directly or by copying some value stored in one of the memory locations of the computer. Along with the STA instruction, it is probably the most frequently used instruction of the entire 6502 set. Its main uses are for placing specific values into memory; for example,

```
LDA #2
STA $0344
```

and for transferring the contents of one memory location to another; for example,

```
LDA $0620
STA $0344
```

This instruction was thoroughly described in Chapter 4.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number loaded into the accumulator is greater than 127, the Negative flag is set; otherwise, it is reset. If the number loaded into the accumulator is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

The eight addressing modes availabe for the LDA instruction were thoroughly described in Chapter 5. The eight modes are briefly listed below:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	LDA #2	2	2	#2 = >A
Absolute	LDA $3420	4	3	contents of memory $3420 = >A
Zero Page	LDA $F6	3	2	contents of memory $F6 = >A
Zero Page,X	LDA $F6,X	4	2	contents of memory $F6 + X = >A
Absolute,X	LDA $3420,X	4	3	contents of memory $3420 + X = >A

Mode	Instruction	Cycles	Bytes	Meaning
Absolute,Y	LDA $3420,Y	4	3	contents of memory $3420 + Y = > A
Ind. Indir.	LDA ($F6,X)	6	2	contents of addr. at $F6 + X = > A
Indir. Ind.	LDA ($F6),Y	5	2	contents of (address at $F6) + offset Y = > A

LDX Load the X Register

THE INSTRUCTION

The LDX instruction directly loads the X register, and is exactly analogous to the LDA instruction for the accumulator. It allows direct loading of the index register and is frequently used in assembly language programs.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number loaded into the X register is greater than 127, the Negative flag is set; otherwise, it is reset. If the number loaded into the X register is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

LDX utilizes five addressing modes. A summary is given below:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	LDX #2	2	2	LDX #2
Absolute	LDX $3420	4	3	LDX contents of memory $3420
Zero Page	LDX $F6	3	2	LDX contents of memory $F6
Zero Page,Y	LDX $F6,Y	4	2	LDX contents of memory $F6 + Y
Absolute,Y	LDX $3420,Y	4	3	LDX contents of memory $3420 + Y

LDY Load the Y Register

THE INSTRUCTION

The LDY instruction, like the LDA and LDX instructions, allows direct loading of the 6502. In this case, the load is into the Y register, but in all other respects LDY is identical to both the LDX and LDA instructions.

EFFECTS ON THE PROCESSOR STATUS REGISTER

As with the LDX instruction, if the number loaded into the Y register by LDY is greater than 127, the Negative flag is set; otherwise, it is reset. If the number loaded is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

Five addressing modes are available for the LDY instruction, as outlined here:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	LDY #2	2	2	LDY #2
Absolute	LDY $3420	4	3	LDY contents of memory $3420
Zero Page	LDY $F6	3	2	LDY contents of memory $F6
Zero Page,X	LDY $F6,X	4	2	LDY contents of memory $F6 + X
Absolute,X	LDY $3420,X	4	3	LDY contents of memory $3420 + X

LSR Logical Shift Right

THE INSTRUCTION

This instruction is the exact opposite of ASL. The LSR instruction forces the most significant bit (bit 7) of a number to zero and rotates each bit down 1 position, with the least significant bit (bit 0) ending up in the Carry bit.

```
                76543210 Carry
before: 0 => 10110101 => 0
after:         01011010    1
```

We can see the shift in the bits within the number and the transfer of the zero into the high bit, as well as the transfer of the low bit into the Carry.

Remember that an ASL instruction causes a number to double its value. Since each bit in a binary number is exactly one-half the value of its left-hand neighbor, the LSR instruction divides a number by 2, and the Carry bit represents the remainder of the division. In the example above, these are the 2 bytes before and after the LSR:

```
#%10110101 = #181
#%01011010 = #90 with C=1
```

We can see that the division worked as we expected.

> **CAUTION:** If we are using signed arithmetic, then the first byte in the example is not 181, but rather $-(255-181) = -74$, and we all know that 90 is not half of -74. To divide a negative number by 2, we have to remember that it is negative, then convert it to its positive counterpart, divide it by 2, and then convert it back to a negative number. Whew! Let's see how to do this:

```
LDA #$FE  ;-2
EOR #$FF  ;complement it
CLC       ;before adding
ADC #1    ;now it's +2
LSR A     ;divide by two
EOR #$FF  ;complement again
CLC       ;before adding
ADC #1    ;as above
STA ...   ;the answer, -1
```

From this example, we can see that to interconvert positive and negative numbers, we need only to EOR the number with #$FF, and add 1 to the result.

EFFECTS ON THE PROCESSOR STATUS REGISTER

Since the high bit of the number being addressed is always forced to zero, the Negative flag is always reset by this operation. If the result of the LSR instruction is equal to zero, the Zero flag is set; otherwise, it is reset. Finally, the Carry flag is set if the least significant bit of the original number was a 1, and it is reset if this bit was a zero.

ADDRESSING MODES

Five addressing modes are available for the LSR instruction:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	LSR $3420	6	3	LSR contents of memory $3420
Zero Page	LSR $F6	5	2	LSR contents of memory $F6
Zero Page,X	LSR $F6,X	6	2	LSR contents of memory $F6 + X
Absolute,X	LSR $3420,X	7	3	LSR contents of memory $3420 + X
Accumulator	LSR A	2	1	LSR contents of accumulator

NOP No Operation

THE INSTRUCTION

The NOP instruction acts as you might expect from its name; it does nothing! So why have it at all? The NOP instruction can be used to hold space for modifying instructions in a program, or in debugging a program, to eliminate an instruction without having to change the location in memory of all succeeding instructions.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

As can be deduced from its operation, the only addressing mode available for the NOP instruction is the implied mode. It takes 1 byte of memory and requires 2 machine cycles to execute.

ORA Or Memory with the Accumulator

THE INSTRUCTION

The ORA instruction is the last of the three logical instructions of the 6502. The first two are the AND and EOR instructions. The ORA instruction compares two numbers bit by bit, and if a bit is set to 1 in either or both numbers, that bit will also be set to 1 in the resulting number. Let's look at an example:

Number 1:		#%10100101
Number 2:		#%01101100
ORA result:	=	#%11101101

The primary use of ORA is to set a particular bit of a number to 1. For instance, if you have a number in memory location $4235 and you need to use it with the least significant bit (bit 0) set to 1, you simply load the accumulator with the number 1 and ORA with memory location $4235. The accumulator will then contain the number which was in memory location $4235, with its least significant bit set equal to 1.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number residing in the accumulator following the ORA instruction is equal to zero, the Zero flag will be set; otherwise, it will be reset. If the resulting number is greater than 127, the Negative flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The ORA instruction utilizes the following eight addressing modes:

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	ORA #2	2	2	A OR #2
Absolute	ORA $3420	4	3	A OR contents of memory $3420
Zero Page	ORA $F6	3	2	A OR contents of memory $F6
Zero Page,X	ORA $F6,X	4	2	A OR contents of memory $F6 + X
Absolute,X	ORA $3420,X	4	3	A OR contents of memory $3420 + X
Absolute,Y	ORA $3420,Y	4	3	A OR contents of memory $3420 + Y
Ind. Indir.	ORA ($F6,X)	6	2	A OR contents of addr. at $F6 + X
Indir. Ind.	ORA ($F6),Y	5	2	A OR contents (address at $F6) + offset Y

PHA Push the Accumulator onto the Stack

THE INSTRUCTION

In assembly language programming, we generally have several places in which to store a value temporarily. We can place it into some reserved memory location or place it in the X or Y registers or push it onto the stack. Of these methods, the only one which won't disturb any other stored information (as the TAY or TAX instructions might), is the PHA instruction, which will push the number onto the stack. However, great caution must be exercised when using the stack to store information. Remember that the stack is used to hold return addresses so that the JSR instruction will know where to return following the completion of the subroutine. Pushing extraneous numbers onto the stack can result in computer crashes unless you take care not to interfere with the return addresses, since the 6502 will try to return to a virtually random address; the odds of finding a valid instruction at such an address are vanishingly small.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The only addressing mode available for the PHA instruction is the Implied mode. The instruction requires only 1 byte of memory and 3 machine cycles to execute.

PHP Push the Processor Status Register onto the Stack

THE INSTRUCTION

The PHP instruction takes the byte containing the flags of the processor status register and pushes it onto the stack. Its purpose is to save the contents of the processor status register for some future operation while intermediate steps are being processed. The cautions for using PHP are similar to those for the PHA instruction: be sure you don't interfere with information that would normally be placed onto the stack, such as return addresses for subroutine operations.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

As might be expected, the only addressing mode for the PHP instruction is the Implied mode. The PHP instruction is a 1-byte instruction and requires 3 machine cycles to execute.

PLA Pull the Accumulator from the Stack

THE INSTRUCTION

This instruction is the counterpart to the PHA instruction. Obviously, if we have a way to push the value stored in the accumulator onto the stack, we should also have a way to get it back again. The PLA instruction removes the top value from the stack and places it into the accumulator for further use.

We will discuss one important use of PLA, but first we need to know a little about machine language subroutines that are to be used in BASIC. Let's suppose that since ATARI BASIC has no true AND function, we want to write a machine language subroutine that will AND two numbers together and return the answer to BASIC. The first problem we face is how to get the two numbers to our machine language routine. There are two ways to do this.

The first method is universal and can be used on most microcomputers. First, we calculate the high and low bytes of the two numbers and then POKE each of these bytes into memory. The machine language subroutine then accesses these memory locations to obtain the numbers, and, after ANDing them together, places the answer in memory, where it can be accessed by your BASIC program. The BASIC program looks like this:

```
10 HIGHP=INT(P/256):REM GETS HIGH BYTE OF P
20 LOWP=P-256*HIGHP:REM GETS LOW BYTE OF P
30 HIGHQ=INT(Q/256):REM GETS HIGH BYTE OF Q
40 LOWQ=Q-256*HIGHQ:REM GETS LOW BYTE OF Q
50 POKE ADDR1,LOWP:REM PUTS LOWP INTO MEMORY
60 POKE ADDR2,HIGHP:REM PUTS HIGHP INTO MEMORY
70 POKE ADDR3,LOWQ:REM PUTS LOWQ INTO MEMORY
80 POKE ADDR4,HIGHQ:REM PUTS HIGHQ INTO MEMORY
90 X=USR(1536):REM ACCESSES MACHINE LANGUAGE SUBROUTINE
100 LOWANS=PEEK(ADDR4):REM GET LOW BYTE OF ANSWER
110 HIGHANS=PEEK(ADDR5):REM GET HIGH BYTE OF ANSWER
120 ANSWER=LOANS+256*HIGHANS:REM CONSTRUCT ANSWER
```

Although this method works, it's a bit clumsy. But for a number of microcomputers on the market, this is the only method available.

However, your ATARI has a much more elegant and simple solution to the problem.

The solution involves passing parameters from BASIC to machine language and back again. These parameters can be numbers, addresses of strings, or virtually any type of constant or variable in your BASIC program. The method for passing parameters to a machine language program is simply to list the parameters to be passed after the address of the machine language routine in the USR call, separating the parameters by commas. Here is the program to do this:

```
10 ANSWER=USR(1536,P,Q)
```

Yes! Only one line! How do we do this?

Your ATARI takes P and Q, breaks them down into high and low bytes for you, and places them onto the stack, where they can be retrieved by your machine language program using the PLA instruction. It's also possible, by storing the answer obtained by your machine language routine into $D4 and $D5 (low byte first), to have the variable ANSWER automatically contain the right answer after line 10 is executed.

There is one crucial detail concerning this use of parameter passing. Since the ATARI allows parameters to be passed in a USR call, it also tells the machine language routine how many parameters are being passed. It does this even if no parameters are being passed! This information is automatically pushed onto the stack as a single byte as soon as the BASIC line is executed. In the case of line 10 above, therefore, the stack will have the number 2 pushed onto it before the high and low bytes of P and Q are placed there. This byte then fouls up the return address, so the machine language program would fail to return to the right place in the BASIC program. Your computer will crash. In all likelihood, you'll have to turn the power off and on again, losing your program. How can we prevent this?

The answer is as simple as it is obvious. Just begin every machine language subroutine you write for a BASIC program with a PLA instruction. PLA will get rid of this extra byte on the stack, and then you can proceed with the remainder of your machine language routine. This byte that PLA pulled from the stack into the accumulator can be used as a check on your BASIC program, if you desire. For instance, we already know that the number in this example should be 2. If it's not, someone made a mistake in the BASIC program. We already know that our machine language routine will fail if two parameters are not passed to it. Therefore, we could build code into our machine language routine to check the first byte pulled into the accumulator. If it is not a 2, the program could branch to some routine which notifies the user of the error by printing an error message or passing an error code back into ANSWER or using another method.

In any case, it is critical to remember that when you write machine language subroutines for use in BASIC programs, you must include the extra PLA to remove the number of parameters from the stack or your computer will crash.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number which the PLA instruction pulls from the stack is greater than 127, the Negative flag will be set; otherwise, it will be reset. If the number pulled from the stack is equal to zero, the Zero flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The only addressing mode available for the PLA instruction is the Implied mode, requiring only 1 byte of memory and 4 machine cycles to execute.

PLP Pull the Processor Status Register from the Stack

THE INSTRUCTION

PLP reverses the PHP instruction by removing the top byte from the stack and placing it into the processor status register. PLP is used to restore things to the way they were at the time the PHP instruction was executed.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The PLP instruction affects all of the flags in the processor status register, since they will all be changed to the values contained in the byte pulled from the stack.

ADDRESSING MODES

The only addressing mode available for the PLP instruction is the Implied mode. PLP is a 1-byte instruction and requires 4 machine cycles to execute.

ROL Rotate Left

THE INSTRUCTION

ROL is similar to the ASL and LSR instructions, but with one significant difference. When those two instructions are executed, the rotation of the bits forces a zero into the high or low bit, respectively. The ROL instruction is a true rotation, in which the Carry bit is placed into bit zero of the number, each bit of the number is moved one position to the left, and the most significant bit of the number is rotated into the Carry bit. This can be shown pictorially as follows:

```
               76543210        C
before:        10010101   <=   0
             C 76543210
after:       1 00101010
```

Note that in contrast to the ASL and LSR instructions, ROL doesn't change the actual bits themselves; it changes only their positions within the number. For example, if we write a program which has eight consecutive ROL instructions, we return to the same number with which we began. Using eight consecutive ASL or LSR instructions would give us an answer of zero, since for each of the eight instructions, one more bit is set to zero.

Each bit of the original number is moved one position to the left following a ROL instruction, so if the Carry bit is initially zero, the original number is doubled each time this instruction is used. The same cautions discussed in the section on the ASL instruction also apply to the use of the ROL instruction.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The ROL instruction sets the Carry bit equal to the value of the most significant bit of the original number. If the answer is equal to zero, the Zero bit will be set; otherwise, it will be reset. Finally, if the answer is greater than 127 (which will happen if bit 6 of the original number is a 1), the Negative flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The ROL instruction uses the same five addressing modes as do the ASL and LSR instructions:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	ROL $3420	6	3	ROL contents of memory $3420
Zero Page	ROL $F6	5	2	ROL contents of memory $F6
Zero Page,X	ROL $F6,X	6	2	ROL contents of memory $F6 + X
Absolute,X	ROL $3420,X	7	3	ROL contents of memory $3420 + X
Accumulator	ROL A	2	1	ROL contents of accumulator

ROR Rotate Right

THE INSTRUCTION

As you might guess, the ROR and ROL instructions are exact opposites. ROR performs a rotation to the right, dividing a number by 2 if the Carry bit was initially zero. Apply the same cautions that were discussed for the LSR instruction. ROR can be represented pictorially as follows:

```
          C     76543210
before:   1  => 01101100
                76543210 C
after:          10110110 0
```

Note that the rotation of the bits is to the right and the original Carry bit is transferred into bit 7 of the answer. Bit 0 of the starting number is transferred to the Carry bit following the operation.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The Carry bit will be set to the value of the least significant bit (bit 0) of the number being rotated. If the resulting number is equal to zero, the Zero flag will be set; otherwise, it will be reset. If the resulting number is greater than 127 (which will happen if the Carry bit had been set prior to the ROR), the Negative flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The five addressing modes available for the ROR instruction are the same as those just discussed for the ROL instruction and are outlined below:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	ROR $3420	6	3	ROR contents of memory $3420
Zero Page	ROR $F6	5	2	ROR contents of memory $F6
Zero Page,X	ROR $F6,X	6	2	ROR contents of memory $F6 + X
Absolute,X	ROR $3420,X	7	3	ROR contents of memory $3420 + X
Accumulator	ROR $3420,Y	2	1	ROR contents of memory $3420 + Y

RTI Return from Interrupt

THE INSTRUCTION

As we have already discussed briefly, the ATARI makes frequent use of interrupt routines. These divert programs from their normal flow to a new section which performs a particular funtion, and then return the program flow back to the point at which the interrupt occurred. The RTI instruction is provided in the 6502 instruction set to accomplish this return to normal flow.

When an interrupt occurs, the 6502 transfers the contents of the processor status register and the program counter onto the stack. When an RTI instruction is encountered, the microprocessor restores these registers from the stack, thus returning to the exact state of the 6502 prior to the interrupt and allowing program flow to resume.

EFFECTS ON THE PROCESSOR STATUS REGISTER

All flags in the processor status register may be changed by the RTI instruction, since the entire register is renewed by pulling the original value off the stack and returning it here.

ADDRESSING MODES

The RTI instruction uses only the Implied mode of addressing, requiring 1 byte and 6 machine cycles to execute.

RTS Return from Subroutine

THE INSTRUCTION

The RTS instruction in assembly language programming is analogous to the RETURN command of BASIC: it returns to the next statement following the jump to the subroutine. The instruction causes the program counter to be reloaded with the return address, which is taken from the stack where it was placed by the JSR command. In the discussions of the PHA and PHP instructions, we noted the problem which can arise if other numbers have inadvertently been placed on the stack: when the return address is taken, it is wrong and the program dies.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The RTS instruction uses only the Implied addressing mode, requiring only 1 byte of memory and 6 machine cycles to execute.

SBC Subtract with Carry

THE INSTRUCTION

SBC is the only subtraction instruction of the 6502. It allows you to determine the difference between two numbers by loading the first into the accumulator and subtracting the second. For instance, if we want to perform the subtraction

8 – 6 = ?

we can write the following assembly language program to do so:

```
LDA #8
SBC #6
```

The answer, 2, remains in the accumulator until needed.

In decimal subtraction, we learned in school that if we don't have a sufficiently large number in the ones column to perform the subtraction, we can borrow 1 ten from the tens column, convert it to 10 ones, add this to the number in the ones column, and perform the subtraction. For instance, when we subtract

24 – 9 = ?

we know that we cannot subtract 9 from 4. We borrow 1 ten from the tens column, which becomes 10 ones, and add this to the 4 we began with, giving us 14 ones. Now we can subtract 9 from 14, leaving 5 in the ones column and obtain the correct answer, 15.

The 6502 performs subtraction in the same way, but it borrows from the Carry bit. For this reason, we must always be certain that the Carry bit is set (equal to 1) before doing a subtraction so that if we need to borrow, we'll have something to borrow. We can be sure that the Carry bit is set by using the SEC instruction which sets the Carry bit. Now, to do a more complicated subtraction, the assembly language code might look like this:

```
SEC      ;be sure C is set
LDA #24  ;1st number
SBC #26  ;2nd number
?        ;answer now in accumulator
```

What is the answer at the "?"? We set the Carry bit before we started, and it's obvious that we needed to borrow before we could perform the subtraction. Therefore, it's apparent that the Carry bit following this subtraction will be zero. When it is used for borrowing, the Carry bit has a value of 256; we began with the number 256 + 24 = 280 and we subtracted 26, leaving an answer of 254.

Using the SBC instruction, we can subtract any number from another. Note that here we have confined our examples to numbers which can be expressed in a single byte. Double-precision arithmetic was used in several examples in Chapters 7 to 10.

EFFECTS ON THE PROCESSOR STATUS REGISTER

As described above, the Carry bit will be reset if a borrow is necessary to perform the subtraction. The Negative flag is set if the answer is greater than 127; otherwise, it is reset. The Zero flag is set if the answer is equal to zero; otherwise, it is reset. The Overflow flag is set when the answer is larger than plus or minus 127; otherwise, it is reset.

ADDRESSING MODES

The SBC instruction uses the same eight addressing modes as the LDA instruction. Please refer to Chapter 5 for details of these modes.

Mode	Instruction	Cycles	Bytes	Meaning
Immediate	SBC #2	2	2	A+#2
Absolute	SBC $3420	4	3	A-contents of memory $3420
Zero Page	SBC $F6	3	2	A-contents of memory $F6
Zero Page,X	SBC $F6,X	4	2	A-contents of memory $F6+X
Absolute,X	SBC $3420,X	4	3	A-contents of memory $3420+X
Absolute,Y	SBC $3420,Y	4	3	A-contents of memory $3420+Y
Ind. Indir.	SBC ($F6,X)	6	2	A-contents of addr. at $F6+X
Indir. Ind.	SBC ($F6),Y	5	2	A-contents of (address at $F6) +offset Y

SEC Set the Carry Bit

THE INSTRUCTION

This instruction is used whenever it is necessary to set the Carry bit to 1. The primary use of SEC is prior to subtractions, as described for SBC. Another use of SEC is prior to a rotate command, when you want to force the Carry bit to 1 before the rotation.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The only effect of the SEC instruction on the processor status register is to set the Carry bit unconditionally.

ADDRESSING MODES

The only addressing mode for the SEC instruction is the Implied mode. It is a 1-byte instruction and requires 2 machine cycles to execute.

SED Set the Decimal Mode

THE INSTRUCTION

As discussed in the sections on the CLD and ADC instructions, the 6502 can operate either in the binary or the decimal mode. To set it in the binary mode, we use the CLD instruction, and to set it in the decimal mode, we use the SED instruction. Note that all additions and subtractions following SED will be in the decimal mode, until cleared by the CLD instruction.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The SED instruction unconditionally sets the Decimal flag equal to 1. It has no other effects.

ADDRESSING MODES

The only addressing mode used by the SED instruction is the Implied mode, using 1 byte of memory and 2 machine cycles to execute.

SEI Set the Interrupt Flag

THE INSTRUCTION

As mentioned previously, setting the interrupt flag will prevent maskable interrupts such as display list and vertical blank interrupts from occurring. There are times when we would like to write our own **interrupt handler**, a program which the computer will execute whenever an interrupt occurs. Before we direct the computer to our routine, it's good programming practice to set the interrupt flag, direct the 6502 to our routine's location by setting the appropriate vector (discussed in Chapter 8), and then clear the interrupt flag to resume normal operations. This is how we prevent an interrupt from occurring while we are changing the address of the interrupt vector. If such an interrupt occurred when we were halfway through the change, the computer would no doubt crash. The SEI instruction prevents this from happening.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The only effect of the SEI instruction is to unconditionally set the Interrupt flag to 1.

ADDRESSING MODES

Like the two previous instructions, the SEI instruction uses only the Implied addressing mode, and takes 1 byte of memory and 2 machine cycles to execute.

STA Store the Accumulator in Memory

THE INSTRUCTION

Just as the LDA instruction can load the accumulator from any memory location, the STA instruction can store whatever value is in the accumulator into any memory location. Note that this instruction does not affect the value stored in the accumulator. It just copies this value into some memory location for storage. We know how to move a value from one place in memory to another, as follows:

```
LDA $2468  ;load from location 1
STA $1357  ;and store in location 2
```

In fact, we can now get fairly sophisticated and transfer a whole block of memory from one place to another:

```
     LDY #$50       ;set up number of bytes to move
LOOP LDA $5678,Y    ;get first byte from $56C8
     STA $4567,Y    ;deposit it in $45B7
     DEY            ;decrement counter
     BNE LOOP       ;if counter>0 then loop
     .              ;gets here only when done
```

Look particularly at the LDA and STA instructions. Both use the Absolute,Y addressing mode, which allows Y to act not only as the loop counter, but also as the offset from the base addresses from which to load, and to which to transfer. The first byte transferred by this routine is the highest byte of the block, and the routine moves down through memory until the last byte transferred is the one from $5678 to $4567. After that transfer, when we decrement the Y register again, it will equal zero. Therefore, the branch back to LOOP will not be taken, since we loop to LOOP only if the Zero flag is not equal to 1 at this point.

Let's look briefly at the speed of such a routine. We have moved 80 bytes of memory (remember, #$50 = #80). First we loaded the Y register using the Immediate mode, which takes 2 machine cycles. Then we go through the loop 80 times. Each loop consists of one LDA and one STA, both in the Absolute,Y addressing mode, one DEY, and one BNE instruction. If we add the cycles for the loop, we get

LDA	=	4
STA	=	4
DEY	=	2
BNE	=	2
Total	=	12

Multiplying by 80 loops yields 960 machine cycles, and adding the 2 for the LDY instruction gives a total time of 962 machine cycles. Since each cycle in your ATARI takes 0.56 microseconds, the total elapsed time to move 80 bytes of memory from one location to another was 538.72 microseconds. In BASIC, using a program similar to this

```
10 FOR I=1 TO 80
20 POKE ADDR1+I,PEEK(ADDR2+I)
30 NEXT I
```

to accomplish the same end requires 1.25 seconds! Using a machine language routine to accomplish this task increased the speed 2329-fold! Other examples of transferring blocks of memory can be found in Chapter 7.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The STA instruction uses seven of the eight addressing modes available to its counterpart, the LDA instruction. Addressing modes are fully explained in Chapter 5 and are merely outlined here:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	STA $3420	4	3	STA A into memory $3420
Zero Page	STA $F6	3	2	STA A into memory $F6
Zero Page,X	STA $F6,X	4	2	STA A into memory $F6+X
Absolute,X	STA $3420,X	4	3	STA A into memory $3420+X
Absolute,Y	STA $3420,Y	4	3	STA A into memory $3420+Y
Ind. Indir.	STA ($F6,X)	6	2	STA A into addr. at $F6+X
Indir. Ind.	STA ($F6),Y	5	2	STA A into (address at $F6) + offset Y

STX Store the X Register

THE INSTRUCTION

The STX instruction may be used exactly like the STA instruction, except that the value in the X register, rather than in the accumulator, is stored in memory.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The STX instruction uses three addressing modes as outlined below:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	STX $3420	4	3	STX into memory $3420
Zero Page	STX $F6	3	2	STX into memory $F6
Zero Page,Y	STX $F6,Y	4	2	STX into memory $F6 + Y

STY Store the Y Register

THE INSTRUCTION

STY, just like the STA and STX instructions, stores the value contained in a register (this time, the Y register) into memory.

EFFECTS ON THE PROCESSOR STATUS REGISTER

None.

ADDRESSING MODES

The three addressing modes used with the STY instruction are outlined below:

Mode	Instruction	Cycles	Bytes	Meaning
Absolute	STY $3420	4	3	STY into memory $3420
Zero Page	STY $F6	3	2	STY into memory $F6
Zero Page,X	STY $F6,X	4	2	STY into memory $F6 + X

TAX Transfer Accumulator to the X Register

THE INSTRUCTION

TAX is a transfer instruction which copies the value stored in the accumulator into the X register, leaving the accumulator unchanged.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number transfered is greater than 127, the Negative flag is set; otherwise, it is reset. If the number transfered is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

Only the Implied mode is available for the transfer instructions, requiring 1 byte of memory and 2 machine cycles.

TAY Transfer Accumulator to the Y Register

THE INSTRUCTION

This instruction transfers the value in the accumulator into the Y register.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number in the accumulator is greater than 127, the Negative flag will be set; otherwise, it will be reset. If the number is equal to zero, the Zero flag will be set; otherwise, it will be reset.

ADDRESSING MODES

Only the Implied mode, which takes 1 byte of memory and 2 machine cycles to execute, is available for the TAY instruction.

TSX Transfer the Stack Pointer to the X Register

THE INSTRUCTION

The transfer instruction TSX copies the stack pointer into the X register, usually prior to storing it for future use.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the stack pointer was greater than 127 (and it usually is), the Negative flag will be set; otherwise, it will be reset. If the stack pointer was equal to zero (almost never), then the Zero flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The TSX instruction uses only the Implied mode, taking 1 byte of memory and 2 machine cycles.

TXA Transfer the X Register to the Accumulator

THE INSTRUCTION

This is the counterpart to the TAX instruction and transfers a value from the X register to the accumulator without changing the value stored in the X register.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The TXA instruction sets the Negative flag if the transfered number was greater than 127; otherwise, it resets it. If the number transfered was equal to zero, the Zero flag will be set; otherwise, it will be reset.

ADDRESSING MODES

The only addressing mode for the TXA instruction is the Implied mode, using 1 byte of memory and 2 machine cycles to execute.

TXS Transfer the X Register to the Stack Pointer

THE INSTRUCTION

This instruction is most frequently used when you wish to set the stack pointer to some predetermined number. Use TXS with extreme caution! You know what can happen when the stack is messed up. Nothing can upset the stack more than the incorrect use of the TXS instruction, so be very careful in its use.

EFFECTS ON THE PROCESSOR STATUS REGISTER

The TXS instruction has no effect on any of the flags in the processor status register.

ADDRESSING MODES

The only addressing mode for the TXS instruction is the Implied mode, using 1 byte of memory and 2 machine cycles to execute.

TYA Transfer the Y Register to the Accumulator

THE INSTRUCTION

TYA transfers the number stored in the Y register into the accumulator, leaving a copy of it in the Y register. It is the counterpart of the TAY instruction already discussed.

EFFECTS ON THE PROCESSOR STATUS REGISTER

If the number transfered is greater than 127, the Negative flag is set; otherwise, it is reset. If the number is equal to zero, the Zero flag is set; otherwise, it is reset.

ADDRESSING MODES

Like the other transfer instructions, the TYA instruction uses only the Implied addressing mode, requiring 1 byte of memory and 2 machine cycles to execute.

APPENDIX TWO
THE THREE CHARACTER SETS USED IN ATARI COMPUTERS

ATASCII	INTERNAL	DISPLAY	CHARACTER
0	160	64	control-comma
1	191	65	control-A
2	149	66	control-B
3	146	67	control-C
4	186	68	control-D
5	170	69	control-E
6	184	70	control-F
7	189	71	control-G
8	185	72	control-H
9	141	73	control-I
10	129	74	control-J
11	133	75	control-K
12	128	76	control-L
13	165	77	control-M
14	163	78	control-N
15	136	79	control-O
16	138	80	control-P
17	175	81	control-Q
18	168	82	control-R
19	190	83	control-S
20	173	84	control-T
21	139	85	control-U
22	144	86	control-V

ATASCII	INTERNAL	DISPLAY	CHARACTER
23	174	87	control-W
24	150	88	control-X
25	171	89	control-Y
26	151	90	control-Z
27	28	91	escape
28	142	92	control-minus
29	143	93	control-equal
30	134	94	control-plus
31	135	95	control-asterisk
32	33	0	space
33	95	1	!
34	94	2	“
35	90	3	#
36	88	4	$
37	93	5	%
38	91	6	&
39	115	7	‘
40	112	8	(
41	114	9	)
42	7	10	*
43	6	11	+
44	32	12	,
45	14	13	-
46	34	14	.
47	38	15	/
48	50	16	0
49	31	17	1
50	30	18	2
51	26	19	3
52	24	20	4
53	29	21	5
54	27	22	6
55	51	23	7
56	53	24	8
57	48	25	9
58	66	26	:
59	2	27	;
60	54	28	<
61	15	29	=
62	55	30	>
63	102	31	?

ATASCII	INTERNAL	DISPLAY	CHARACTER
64	117	32	@
65	127	33	A
66	85	34	B
67	82	35	C
68	122	36	D
69	106	37	E
70	120	38	F
71	125	39	G
72	121	40	H
73	77	41	I
74	65	42	J
75	69	43	K
76	64	44	L
77	101	45	M
78	99	46	N
79	72	47	O
80	74	48	P
81	111	49	Q
82	104	50	R
83	126	51	S
84	109	52	T
85	75	53	U
86	80	54	V
87	110	55	W
88	86	56	X
89	107	57	Y
90	87	58	Z
91	96	59	[
92	70	60	shift-plus
93	98	61	]
94	71	62	shift-asterisk
95	78	63	shift-minus
96	162	96	control-period
97	63	97	a
98	21	98	b
99	18	99	c
100	58	100	d
101	42	101	e
102	56	102	f
103	61	103	g
104	57	104	h

ATASCII	INTERNAL	DISPLAY	CHARACTER
105	13	105	i
106	1	106	j
107	5	107	k
108	0	108	l
109	37	109	m
110	35	110	n
111	8	111	o
112	10	112	p
113	47	113	q
114	40	114	r
115	62	115	s
116	45	116	t
117	11	117	u
118	16	118	v
119	46	119	w
120	22	120	x
121	43	121	y
122	23	122	z
123	130	123	control-semicolon
124	79	124	shift-equals
125	118	125	shift-clear
126	52	126	delete-backspace
127	44	127	tab
155	12	—-	RETURN
156	116	—-	shift-delete
157	119	—-	shift-insert
253	158	—-	control-2(bell)
254	180	—-	control-delete
255	183	—-	control-insert
—-	60	—-	CAPS/lowercase
—-	39	—-	ATARIkey

APPENDIX THREE
THE ATARI MEMORY MAP

Dec.	Hex.	#	Label	Use of the Location(s)
2	2	2	CASINI	Cassette initialization vector
6	6	1	TRAMSZ	Equals 1 if A cartridge present
7	7	1	TSTDAT	Equals 1 if B cartridge present
10	A	2	DOSVEC	Disk software start vector
12	C	2	DOSINI	Disk boot initialization address
14	E	2	APPMHI	Top of applications memory
16	10	1	POKMSK	POKEY interrupts enabled
18	12	3	RTCLOK	Real-time clock
48	30	1	STATUS	Internal SIO status storage location
54	36	1	CRETRY	# of retries of commands
55	37	1	DRETRY	# of device retries
66	42	1	CRITIC	Critical I/O flag during VBI
73	49	1	ERRNO	Disk I/O error number
77	4D	1	ATRACT	If > 127, screen colors rotate
82	52	1	LMARGN	Left margin of screen
83	53	1	RMARGN	Right margin of screen
84	54	1	ROWCRS	Current cursor row
85	55	2	COLCRS	Current cursor column

Dec.	Hex.	#	Label	Use of the Location(s)
87	57	1	DINDEX	Current screen graphics mode
88	58	2	SAVMSC	Address of screen memory
106	6A	1	RAMTOP	RAM size in pages
128	80	2	LOMEM	BASIC's bottom of memory pointer
130	82	2	VNTP	Address of variable name table
132	84	2	VNTD	End of variable name table + 1
134	86	2	VVTP	Address of variable value table
136	88	2	STMTAB	Address of BASIC statement table
140	8C	2	STARP	String & array table pointer
142	8E	2	RUNSTK	Address of BASIC run-time stack
144	90	2	MEMTOP	Top of BASIC memory
186	BA	2	STOPLN	Line # where program stopped
195	C3	1	ERRSAV	Error code #
201	C9	1	PTABW	Columns between TABs
212	D4	6	FR0	Floating point register 0
224	E0	6	FR1	Floating point register 1
237	ED	1	EEXP	Value of exponent
238	EE	1	NSIGN	Sign of floating point number
239	EF	1	ESIGN	Sign of exponent
241	F1	1	DIGRT	# digits to right of decimal
251	FB	1	DEGFLG	For radians = 0;for degrees = 6
512	200	2	VDSLST	NMI DLI vector
528	210	2	VTIMR1	POKEY timer 1 interrupt vector
530	212	2	VTIMR2	POKEY timer 2 interrupt vector
532	214	2	VTIMR4	POKEY timer 4 interrupt vector
534	216	2	VIMIRQ	IRQ immediate vector
546	222	2	VVBLKI	VBLANK immediate vector
548	224	2	VVBLKD	VBLANK deferred vector

Dec.	Hex.	#	Label	Use of the Location(s)
559	22F	2	SDMCTL	Direct Memory Access enable
560	230	2	SDLSTL	Address of display list
580	244	1	COLDST	If = 0,warmstart;if = 1,coldstart
623	26F	1	GPRIOR	Priority register shadows $D01B
624	270	8	PADDLx	Paddle values-shadow $D200-D207
632	278	4	STICKx	Joystick values-shadow $D300-D301
636	27C	8	PTRIGx	Paddle triggers-shadow $D300-D301
644	284	4	STRIGx	Stick triggers-shadow $D010-D013
656	290	1	TXTROW	Text window cursor row
657	291	2	TXTCOL	Text window cursor column
660	294	2	TXTMSC	Address of text window
694	2B6	1	INVFLG	If = 0,chars. normal;if = 128,inverse
702	2BE	1	SHFLOK	If = 0,lower case;if = 64,upper case
703	2BF	1	BOTSCR	# text rows in text window
704	2C0	4	PCOLRx	Player-missile color
708	2C4	5	COLORx	Playfield color
736	2E0	2	RUNAD	Run address from disk
738	2E2	2	INITAD	Initialization address from disk
741	2E5	2	MEMTOP	Top of free memory
743	2E7	2	MEMLO	Bottom of free memory
752	2F0	1	CRSINH	If = 0,cursor on;if > 0,cursor off
756	2F4	1	CHBAS	Character set base register
763	2FB	1	ATACHR	Stores color for FILL and DRAWTO
764	2FC	1	CH	Stores last character pressed
768	300	16	misc.	Disk control block
794	31A	38	HATABS	Handler table
832	340	128	IOCBx	Input/Output Control Blocks
40954	9FFA	2		B cartridge start address
40958	9FFE	2		B cartridge initialization address

Dec.	Hex.	#	Label	Use of the Location(s)
49146	BFFA	2		A cartridge start address
49150	BFFE	2		A cartridge initialization address
53248	D000	4	HPOSPx	Horizontal position of player x
53252	D004	4	HPOSMx	Horizontal position of missile x
53256	D008	4	SIZEPx	Size of player x;0,1 or 3
53260	D00C	1	SIZEM	Size of all missiles
53266	D012	4	COLPMx	Hardware player color registers
53270	D016	4	COLPFx	Hardware playfield color registers
53274	D01A	1	COLBK	Hardware background color register
53277	D01D	1	GRACTL	Graphics control register
53278	D01E	1	HITCLR	Clears collision registers
53279	D01F	1	CONSOL	3 console buttons
53760	D200	8	AUDxx	Audio frequency and control registers
53768	D208	1	AUDCTL	Audio control
53769	D209	1	STIMER	Start the POKEY timers
53770	D20A	1	RANDOM	Reads a random number 0-255
53774	D20E	1	IRQEN	Interrupt request enable
54272	D400	1	DMACTL	Direct Memory Access control
54276	D404	1	HSCROL	Horizontal scroll enable
54277	D405	1	VSCROL	Vertical scroll enable
54279	D407	1	PMBASE	Address of PMBASE
54281	D409	1	CHBASE	Address of character base
54282	D40A	1	WSYNC	Wait for horizontal synchchronization
54283	D40B	1	VCOUNT	Line being drawn/2
54286	D40E	1	NMIEN	NMI enable
58460	E45C	3	SETVBV	Set VBLANK vectors
58463	E45F	3	SYSVBV	VBLANK stage 1 entry
58466	E462	3	XITVBV	VBLANK exit

Notes: 1. # refers to the length of the address, in bytes
2. x refers to several related addresses; e.g., STICKx

INDEX

Computers

ASSEMBLY LANGUAGE PROGRAMMING FOR THE ATARI COMPUTERS

MARK CHASIN

"Anyone who has programmed in BASIC, or any other language for that matter, can learn to program in assembly language. The ATARI is among the most impressive of all home computers, but many of its special features are not available from BASIC. The examples in this book are given both in assembly language and, wherever possible, also in BASIC programs which incorporate these assembly language routines to perform tasks from BASIC. You'll be able to use these routines immediately in your own programs. Included are such techniques as reading the joysticks, moving players and missiles; input or output to all possible devices such as printers, disk drives, cassette recorders, the screen and more; vertical blank interrupt routines, display list interrupts, fine horizontal and vertical scrolling, sound, graphics; in short, everything you've always heard the ATARI computers were capable of, but had no idea how to program.

"The routines in this book follow the 'rules' established by ATARI for assembly language programmers, so they will work with any ATARI computer, from the earliest 400 to the most advanced 1450XLD, and everything in between. If you've reached the point where BASIC is no longer enough, and you'd like to progress to a language which gives you absolute control over all functions of your remarkable computer, then begin with Chapter 1, and you'll see how easy it is. Who knows, maybe you'll be the one to write the sequel to STAR RAIDERS!"

— — *from the Preface*

MARK CHASIN, Ph.D. is a principal at MMG Micro Software and president of Micros of Monmouth, a local computer club. His interest in computers began 20 years ago when he first learned to program. Dr. Chasin's professional affilations include the American Association for the Advancement of Science, the New York Academy of Sciences, and the Association for Computer Machinery, and he is among the experts included in *Who's Who in Computer Graphics.*

McGraw-Hill

ISBN 0-07-010679-7