

Assembly Lines:

The Book

A Beginner's Guide
to 6502 Programming
on the Apple II

by

Roger Wagner

A SOFTALK BOOK

\$19.95

ASSEMBLY LINES:
THE BOOK

0000

0001

0002

0003

0004

0005

0006

0007

0008

0009

0010

0011

0012

0013

0014

ASSEMBLY LINES:
THE BOOK

A Beginner's Guide
to 6502 Programming
on the Apple II

BY ROGER WAGNER

Softalk Publishing
1982

© 1982 by Softalk Publishing Inc. All rights reserved. No part of this publication may be copied, transmitted, or reproduced in any way including, but not limited to, photocopy, photography, magnetic, or other recording, without prior written permission of the publisher, with the exception of program listings, which may be entered, stored and executed in a computer system, but not reproduced for publication.

Library of Congress Catalog Card Number: 81-85708

Softalk Publishing
11021 Magnolia Blvd.
North Hollywood, CA 91601

Design by Kurt A. Wahlner Printed in the United States of America

To My Father

Table of Contents

INTRODUCTION	ix
1 APPLE'S ARCHITECTURE	1
6502 Operation. Memory Locations. Hexadecimal Notation.	
2 ASSEMBLERS	13
General Discussion. Source Code. Object Code. Source Code Fields. Pseudo Opcodes. Load/Store Opcodes.	
3 LOOPS and COUNTERS	23
Binary Numbers. The Status Register. Incrementing and Decrementing. Looping with BNE.	
4 LOOPS, BRANCHES, COUT, and PADDLES	31
Looping with BEQ. Branch Offsets and Reverse Branches. Screen Output using COUT. Reading a Game Paddle. Transfer Commands.	
5 I/O ROUTINES using MONITOR and KEYBOARDS	41
Review of Concepts. Compare Commands and Carry Flag. Using Monitor Programs for I/O Routines. Reading Data from Keyboard.	
6 ADDRESSING MODES	51
Immediate, Absolute, Zero Page, Implicit/Implied, and Relative Addressing Modes. Indexed Addressing. Storing Pure Data.	
7 SOUND GENERATION ROUTINES	61
Delays. Altering Program Length. Delay Value in Memory. Delay from Keyboard and Paddles.	
8 THE STACK	71
LIFO. Stack Pointer. PHA. PLA. Stack Storage Capacity.	
9 ADDITION and SUBTRACTION	75
Binary Numbers. ADC. Clearing the Carry. Two-Byte Addition. SBC. Setting Carry for Subtraction. Positive and Negative Numbers. Ones' Complement. Twos' Complement. Sign Flag.	

10	DOS and DISK ACCESS	89
	Disk Access. Overview of DOS. Diskette Organization. Modifying Access Utilities.	
11	SHIFT OPERATORS and LOGICAL OPERATORS	103
	Shift Operators. Logical Operators. AND. Operational vs. Processing Modes. Flow of Control. Inverse Flag. Masking. Inclusive OR. Exclusive OR.	
12	I/O ROUTINES	121
	Print Routines: Data Type; Special Type. Manipulating the Stack Return Address. Input Routines: Binary Input; Combination Applesoft/Assembly Language.	
13	READING/WRITING FILES on DISK	129
	BLOAD/BSAVE. Name File Program using String and Single Key Input, Print, and DOS Command Routines. Text Files. OPEN/READ, OPEN/WRITE. Simulating Program Execution: LANG, CURLIN, Memory Location \$33.	
14	SPECIAL PROGRAMMING TECHNIQUES	145
	Relocatable vs. Nonrelocatable Code. JMP Commands. Determining Program Location. JSR Simulations. Self-Modifying Code. Indirect Jumps.	
	Appendix A	163
	Contest.	
	Appendix B	171
	Assembly Language Commands: Description and Uses.	
	Appendix C	243
	6502 Instruction Set.	
	Appendix D	259
	Monitor Subroutines.	
	Appendix E	263
	ASCII Code and Text Screen Charts.	
	Index	267

2881

2882

2883

2884

2885

2886

2887

2888

2889

2890

2891

2892

2893

2894

2895



INTRODUCTION

One often gets the impression that programming in assembly language is some very difficult and obscure technique used only by *those advanced programmers*. As it happens, assembly language is merely different, and if you have successfully used Integer or Applesoft Basic to do some programming, there's no reason why you can't use assembly language to your advantage in your own programs.

This book will take a rather unorthodox approach to explaining assembly programming. Because you are presumably somewhat familiar with Basic, we will draw many parallels between various assembly language techniques and their Basic counterparts. An important factor in learning anything new is a familiar framework in which to fit the new information. Your knowledge of Basic will provide that framework.

I will also try to describe initially only those technical details of the microprocessor operations that are needed to accomplish our immediate goals. The rest will be filled in as we move to more involved techniques.

This book does not attempt to cover every aspect of machine language programming. It does, however, provide the necessary information and guidance to allow even a somewhat inexperienced person to learn machine language in a minimum of time. You should find the text and examples quite readable, without being overwhelmed by technical jargon or too much material being presented at once.

I'd like to take this opportunity to briefly mention a few of

my own programming philosophies. Writing programs to do a given task is essentially an exercise in problem solving. Problem solving is in fact a subject in itself. No matter what your programming goal is, it will always involve solving some particular aspect that, at that moment, you don't really know how to solve. The most important part is that, if you keep at it, you eventually will get the solution.

One of the key elements in this process, I believe, and the particular point to stress now, is that it is important to be a *tool user*. Programming in any language consists of using the various commands and functions available to you in that language and of putting them all together in a more complex and functioning unit. If you are not familiar with the options you have at any given moment—that is, your tools—the problem-solving process is immensely more difficult.

My intent in this book is to present in an organized way the various operations available in assembly language and how they can be combined to accomplish simple objectives. The more familiar you are with these elements, the easier it will be to solve a particular programming problem.

You may wish to keep your own list of the machine language commands and their functions as we go along. A list of these commands is included in appendix C, but I think you'll agree that by taking the time to write each one down as you learn it, along with your own personal explanation of what it does, you will create a much stronger image in your mind of that particular operation.

You may wish to supplement this book with other books on 6502 programming. Recommended books include:

Randy Hyde, *Using 6502 Assembly Language* (Northridge, CA:

DataMost, 1981); \$19.95.

Don Inman and Kurt Inman, *Apple Machine Language* (Reston, VA: Reston Publishing, 1981); \$12.95.

Lance A. Leventhal, *6502 Assembly Language* (Berkeley: Osborne/McGraw-Hill, 1979); \$16.99.

Rodnay Zaks, *Programming the 6502* (Berkeley: Sybex, 1981); \$13.95.

There are undoubtedly others that are also available, and you should consider your own tastes when selecting which ones seem most appropriate to your own learning style.

An additional concern for a book like this is which assembler to use. (An assembler is an editor-like utility for creating machine language programs. If you're vague on this check chapter two for more information.) Although I'm somewhat biased, my favorite assembler is the one available from Southwestern Data Systems called *Merlin*. It not only contains a good assembler, but also a number of additional utilities and files of interest. *Merlin* is not required, however, as the examples given are written to be compatible with most of the assemblers currently available. These include the *Apple DOS Tool Kit*, *TED II*, the *S.C. Assembler*, and many others.

Also available from Southwestern Data Systems is a utility called *Munch-A-Bug* (M.A.B.) which allows a person to easily trace and *de-bug* programs, a process which can be of tremendous help. M.A.B. also includes its own mini-assembler which can be used for the beginning listings provided in this book.

In terms of hardware, any Apple II or Apple II Plus should be more than adequate for your needs and no additional hardware is required. Disk access is discussed in several chapters, but is otherwise not a concern throughout the remainder of the book.

One warning before you start into the subject of machine language programming. As with any nontrivial endeavor, many people sell themselves short because of what I call the *instant expert myth*. How many people hear someone play a piano well, and say, "My, what a beautiful thing. I think I'll get one and learn how to play myself!" They then spend a substantial amount of money, sit down, and press a few keys. Surprise! To their great disappointment, the Moonlight Sonata does not magically flow from their fingers! They usually then become immediately discouraged and never pursue the area further, turning something that could give them tremendous pleasure into an expensive means of support for a flower vase.

I've seen this same effect in almost every area of human activity. If what you wanted was the Moonlight Sonata, a record will produce the sound you desire. People *know* that it takes talent (talent = 99% practice = 99% time) to play well, but are then disappointed when they can't sit down and perform like an expert immediately.

One of the great secrets to learning anything is to be satisfied with minor learning steps. *Playing* the Apple is in many ways

much easier than learning to play a piano, but you should still not expect to sit down and write the world's greatest database in your first evening.

Set yourself some simple and achievable goals. Can you move *one* byte from one memory location to another? If you can you're well on your way to mastering programming. My feeling is that virtually anyone can become better than eighty to ninety percent of his fellow citizens in any area simply because eighty to ninety percent of the other people aren't willing or inclined to spend the necessary time to learn the skill. Reaching the top ninety-nine percent is certainly difficult, but ninety-five percent is surprisingly easy.

This book is written with the intention of providing those simple achievable steps. And surprisingly enough, by the time you finish this book you will have written a simple database of sorts, along with some sound routines, some programs that use paddles and the disk, and a few other nifties as well!

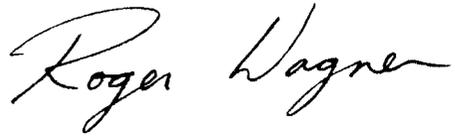
So hang in there and don't expect to be an expert on page five. I will guarantee that by page one hundred you may even surprise yourself as to how easy machine language programming really is.

One final note. I'd like to thank Al Tommervik for his tremendous help and support in this project as both editor and friend, and Greg Voss who provided many insightful suggestions in transforming the monthly series into the book. Also Eric Goetz for his encouragement to never accept less than the best, and his attentive (if not enthusiastic) listening to my various plans over the years.

Last but not least my thanks and sincere thoughts of appreciation to the many people that have shared in my own experiences in computing over the last few years. Whether they were readers of the column, users of my programs, or the wealth of new friends that have entered my life via the Apple, they have made all my efforts more than worthwhile and brought rewards beyond any simple economic gains of an ordinary job.

Alas for anyone who thinks that computers lead to a loss of the humanistic aspects of life. They need only look to the amazing community that has been drawn together from all parts of the world by the Apple to see that friendship and human creativity will always outshine the simple tools we use to express ourselves.

My wish for you, dear reader, is that you receive as much enjoyment from the Apple and programming as I have.

A handwritten signature in black ink that reads "Roger Wagner". The signature is written in a cursive, flowing style with a large initial 'R' and 'W'.

Roger Wagner
Santee, California
December 1, 1981

100

101

102

103

104

105

106

107

108

109

110

111

112

113

114

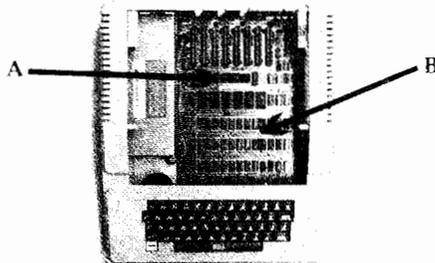


CHAPTER 1

Apple's Architecture

The first area to consider is the general structure of the Apple itself. To help visualize what's going on in there, why not take a look inside. That's right—rip the cover off and see what's in there! Don't be timid—get your nose right down in there and see what you shelled out all those hard-earned bucks for.

Providing you haven't gotten carried away in dismembering your Apple, the inner workings should appear somewhat like those in the photo below.



The main items of interest are the 6502 microprocessor (A) and the banks of memory chips (B). If you're not an electronics whiz, it really doesn't matter. You can take it as a device of magic for all it matters. The memory chips have the capability of storing thousands of individual number values and the 6502 supervises the activities therein. All the rest of the electronic debris within is supplied only to support the memory and the 6502. The circuits allow you to see displays of this data on the screen, and permit the computer to watch the keyboard for your actions.

The screen and keyboard are rather secondary to the nature of the computer and are provided only to make you buy the thing. As far as the Apple is concerned, it could talk to itself perfectly well without either the screen or the keyboard.

6502 Operation

So how does it work? The heart of the system is the 6502 microprocessor. This device operates by scanning through a given range of memory addresses. At each location, it finds some particular value. Depending on what it finds, it executes a given operation. This operation could be adding some numbers, storing a number somewhere, or any of a variety of other tasks. These interpreted values are often called *opcodes*.

In the old days, programmers would ply their trade by loading each opcode, one at a time, into successive memory locations. After a while, someone invented an easier way, using a software device to interpret short abbreviated words called *mnemonics*. A mnemonic is any abbreviated command or code word that sounds somewhat like the word it stands for, such as `STX` for `STore X`. The computer would then figure out which values to use and supervise the storing of these values in consecutive memory locations. This wonder is what is generally called an assembler. It allows us to interact with the computer in a more natural way. In fact, Basic itself can be thought of as an extreme case of the assembler. We just use words like `PRINT` and `INPUT` to describe a whole set of the operations needed to accomplish our desired action.

In some ways, assembly language is even easier than Basic. There are only fifty-five commands to learn, as opposed to more than one hundred in Basic. Machine code runs very fast and generally is more compact in the amount of memory needed to carry out a given operation. These attributes open up many possibilities for programs that would either run too slowly or take up too much room in Basic.

Memory Locations

Probably the most unfamiliar part of dealing with the Apple in regard to machine level operations is the way addresses and numbers in general are treated. Unless you lead an unusually charmed life, at some point in your dealings with your Apple

you have had it abruptly stop what it was doing and show you something like this:

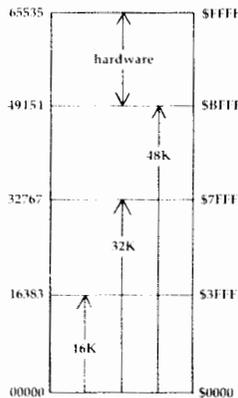
8BF2- A=03 X=9D Y=00 P=36 S=F2

This occurs when some machine level process suddenly encounters a break in its operation, usually from an unwanted modification of memory.

Believe it or not, the Apple is actually trying to tell us something here. Unfortunately, it's rather like being a tourist and having someone shout, "*Alaete quet beideggen!*" at you.¹ It doesn't mean much unless you know the lingo, so to speak. . . .

What has happened is that the Apple has encountered the break we mentioned and, in the process of recovering, has provided us with some information as to where the break occurred and what the status of the computer was at that crucial moment. The message is rather like the last cryptic words from the recently departed.

The leftmost part of the message is of great importance. This is where the break in the operation occurred. Just what do we mean by the word *where*? Remember all that concern about whether you have a 16K, 32K, or 48K Apple? The concern was about the number of usable memory locations in your machine. This idea becomes clearer through the use of a *memory map*, such as the one shown below.



1. "Watch where you're stepping you nerd!" (in case you're not familiar with this particular dialect.)

Inside the Apple are many electronic units that store the numerical values we enter. By numbering these units, we assign each one a unique *address*. This way we can specify any particular unit or memory location, either to inquire about its contents or to alter those contents by storing a new number there.

In the Apple there are a total of 65,536 of these memory locations, called bytes. The chart gives us a way of graphically representing each possible spot in the computer.

When the computer shows us an address, it does not do it in a way similar to the numbers on the left of the memory map, but rather in the fashion of the ones on the right. You may well remark here: "I didn't know BFFF was a number; it sounds more like a wet sneaker. . . ."

Hexadecimal Notation

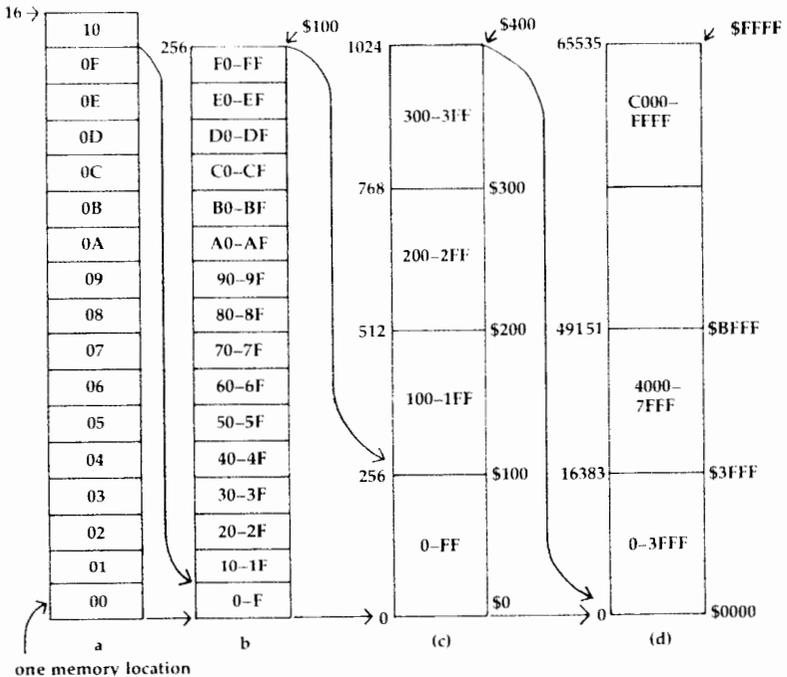
To understand this notation, let's see how the 6502 counts. If we place our byte at the first available location, its address is \$0. The dollar sign is used in this case to show that we are not counting in our familiar decimal notation, but rather in hexadecimal (base sixteen) notation, usually called *hex*, which is how the computer displays and accepts data at the Monitor level.

After byte \$0, successive locations are labeled in the usual pattern up to \$9. At this point the computer uses the characters A through F for the next six locations. The location right after \$F is \$10. This is not to be confused with ten. It represents the decimal number sixteen. The pattern repeats itself as in usual counting with:

\$10, 11, 12, 13 . . . 19, 1A, 1B . . . 1E, 1F, 20

Try not to let this way of counting upset you. The pattern in which a person (or machine) counts is rather arbitrary, and should be judged only on whether it makes accomplishing a task easier or not. The biggest problem for most people is more a matter of having been trained to use names like *one hundred* when they see the numerals 100. How many items this corresponds to really depends more on the conventions we agree to use than on any cosmic decree. To aid in your escape from your possibly narrow view of counting, you may wish to read the diversionary story following this chapter. In any event, it will be sufficient for our purposes to understand that \$1F is as legitimate a number as 31.

The hex number \$FF (255) is the largest value a single byte can hold. A block of 256 bytes (for instance \$0 to \$FF) is often called a *page* of memory. In the figure below, all the addresses from \$0 to \$FF are shown in block b. Four of these blocks together, as in c, make up 1K of memory. As you can see, there are actually 1,024 bytes in 1K. Thus a 48K machine actually has 49,152 bytes of RAM.



Block d shows the Apple's entire range again. If you do not have a full 48K of memory, then the missing range will just appear to hold a constant value (usually \$FF), and you will not be able to store any particular value there.

The range from \$C000 to \$FFFF, an additional 16K, is all reserved for hardware. This means that any data stored in this range is of a permanent nature and cannot be altered by the user. Some areas are actually a physical connection to things like the speaker or game switches. Others, like \$E000 to \$FFFF are filled in by the chips in the machine called ROMS.

ROM stands for Read Only Memory. These chips hold the machine language routines that make up either Applesoft or Integer, depending on whether you have an Apple Plus or the standard model. One of the chips is the Monitor, which is what initializes the Apple when it is first turned on so you can talk to it.

The Monitor can be thought of as a simple supervisor program that keeps the Apple functioning at a rather primitive level of intelligence. It handles basic input and output for the computer, and allows a few simple commands relating to such things as entering, listing, or moving blocks of memory within the Apple. Don't be fooled though. The amount of code required to do just these things is not trivial, and in addition provides us with a ready-made mini-library of routines that we can call from our own programs, as will be shown later in this book.

Apple provides an excellent discussion of the Monitor and its commands and operation within the *Apple II Reference Manual*, currently supplied with all new Apples. You may wish to consult this if you are unsure of the general way in which the Monitor is accessed and used.

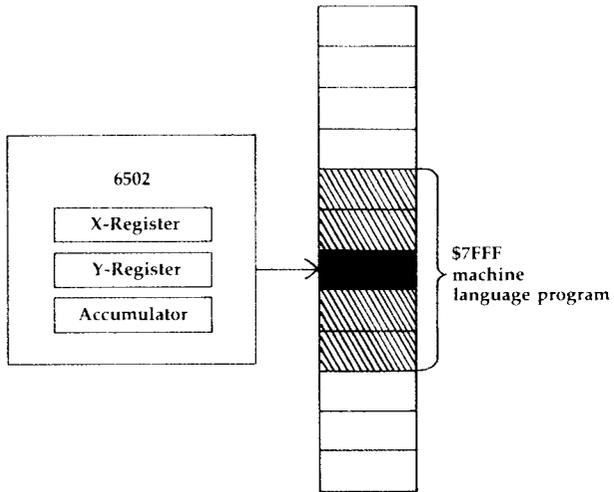
Now that break message should have at least a little meaning.

```
8BF2-   A=03   X=9D   Y=00   P=36   S=F2
```

The 8BF2 is an address in memory. The display indicates that the break actually occurred at the address given minus two (8BF2 - 2 = 8BF0). For reasons that aren't worth going into here, the Monitor always prints out a break address in this plus-two fashion. What about the rest of the message? Consider the next three items:

```
A=03   X=9D   Y=00
```

The 6502, in addition to being able to address the various memory locations in the Apple, has a number of internal *registers*. These are units inside the 6502 itself that can store a given number value, and they are individually addressable in much the same way memory is. The difference is that instead of being given a hexadecimal address, they are called the X-Register, the Y-Register, and the Accumulator. In our error message, we are being told the status of these three registers at the break.



The above figure illustrates what we know so far. The 6502 is a microprocessor chip that has the ability to scan through a given range of memory, which we will generally specify by using hex notation for the addresses. Depending on the values it finds in each location as it scans through, it will perform various operations. As an additional feature to its operation, it has a number of internal registers, specifically the X-Register, the Y-Register, and the Accumulator. Memory-related operations are best done by entering the Monitor level of the Apple (usually with a CALL - 151) and using the various routines available to us.

Exploring the Monitor

It is possible to program the computer manually by entering numbers one at a time into successive memory locations. A program of this sort is called a *machine language* program because the 6502 can directly run the coded program steps. Humans, however, find this type of data difficult to read and are more likely to make mistakes while working with it.

A more convenient method of programming is to assign some kind of code word to each value. The computer will translate this word into the correct number to store in memory. This translation is done by an *assembler*, and programs entered or displayed in this manner are called *assembly language* programs.

As an example, let's look at some data within your Apple,

first in the machine language format and then in the assembly language format. First we must enter the Monitor. Type in:

```
CALL -151
```

This should give you an asterisk (*) as prompt. Now type in:

```
F800.F825
```

This tells the Monitor we want to examine the range of memory from \$F800 to \$F825. The general syntax of the command is:

```
<start address>.<end address>
```

the period being used to separate the two values.

Upon hitting RETURN you should get the following data:

```
F800- 4A 08 20 47 F8 28 A9 0F
F808- 90 02 69 E0 85 2E B1 26
F810- 45 30 25 2E 51 26 91 26
F818- 60 20 00 F8 C4 2C B0 11
F820- C8 20 0E F8 90 F6
*
```

The range I have picked is the very beginning of the Monitor ROM. The data here can be directly read by the 6502, but is very difficult for most humans to make much sense of. This is machine language.

Now type in:

```
F800L
```

This tells the Monitor to give us a disassembly of the next twenty instructions, starting at \$F800. The syntax here is:

```
<start address>L
```

To disassemble means to reverse the process we talked about earlier, taking each number value and translating it into the appropriate code word.

After hitting RETURN you should get:

F800-	4A	LSR	
F801-	08	PHP	
F802-	20 47 F8	JSR	\$F847
F805-	28	PLP	
F806-	A9 0F	LDA	#\$0F
F808-	90 02	BCC	\$F80C
F80A-	69 E0	ADC	#\$E0
F80C-	85 2E	STA	\$2E
F80E-	B1 26	LDA	(\$26),Y
F810-	45 30	EOR	\$30
F812-	25 2E	AND	\$2E
F814-	51 26	EOR	(\$26),Y
F816-	91 26	STA	(\$26),Y
F818-	60	RTS	
F819-	20 00 F8	JSR	\$F800
F81C-	C4 2C	CPY	\$2C
F81E-	B0 11	BCS	\$F831
F820-	C8	INY	
F821-	20 0E F8	JSR	\$F80E
F824-	90 F6	BCC	\$F81C

This is a disassembled listing. Although it probably doesn't do a lot for you right now, I think it's obvious that it is at least more distinctive.

Let's look at it a little more closely. In Basic, line numbers are used to begin each set of statements. They're particularly handy when you want to do a GOTO or GOSUB to some other part of the program. In machine language, the addresses themselves take the place of the line numbers. In our example, the column of numbers on the far left are the addresses at which each operation is found. To the right of each address are one to three hex values, which are number values stored in successive addresses. These are the opcodes with their accompanying *operands*.

At \$F802, for instance, is the opcode \$20. Remember, the dollar sign is used to show we are using base sixteen. \$20 is the opcode for the command JSR. All mnemonics are made up of three letters. In this case, JSR stands for Jump to SubRoutine and is rather like a GOSUB in Basic. The next two numbers, \$47 and \$F8, comprise the operand, that is, the number that the opcode is to use in its operation. To the right we see that these numbers

give \$F847 as the object of the JSR.² Continuing with our analogy, what would be a GOSUB 1000 in Basic appears as a JSR \$F847 in assembly language. The command JSR \$F847 will jump to the subroutine at \$F847 and return when done.

You've just learned your first word of assembly language: JSR! Looking through the listing, you can see several of these. The first one goes to some routine outside the listing. What about the other two JSR commands? You should be able to see that they reference routines within the listing. The second enters at \$F800, the third at \$F80E.

In Basic, a GOSUB eventually ends with a RETURN. The JSR has an analogous counterpart. Looking at the entry point at \$F80E and what follows, can you find anything that looks like it might be the equivalent of a RETURN? Take the time to find it if you can before reading on.

If you picked the RTS, you're right. RTS stands for return from subroutine. As with a RETURN, when the program reaches the RTS, it returns to where it originally came from. Encountering the RTS at \$F818, program execution would resume at \$F824, if entry was from the JSR \$F80E at \$F821.

You might notice that almost all machine code blocks that you may have used along with Basic programs, such as tone routines, usually end with a \$60 as the last byte. This is the opcode for RTS. In almost any assembly language program you write, you must end with an RTS. This is because, to the computer as a whole, your program is a temporary subroutine of its overall operation.

When your program ends, the RTS lets the Apple return to its original operations of scanning the keyboard and such. When you do a CALL 768 from Basic, for example, you are essentially doing a JSR to that machine routine. The 768 is the decimal value for the address of the start of the routine, equivalent to \$300 in

2. Notice that it takes two bytes to store the value for an address. For example, for the address \$F847, the value "F8" is stored in one byte, and "47" in another. Reading an address is generally a matter of mentally combining the two bytes.

The byte representing the left-hand portion of the number is often called the *high-order byte*; the byte representing the right-hand portion is called the *low-order byte*.

It is important to realize that the two bytes that make up an address are almost always *reversed* in regards to what you might normally expect. That is to say that in an address byte-pair, the low-order byte always comes first, immediately followed by the high-order byte. This means that when examining raw memory, you must mentally reverse the byte to determine the address stored. Fortunately when using the L command, the disassembler does this for you.

hexadecimal. At the end of that routine, the RTS returns you to your Basic program to let it continue with the next statement.

It's Culture That Counts

Many people have remarked that our choice of ten as a number base is related to the fact that we have ten fingers on our hands. One can only guess how a different set of circumstances would have profoundly changed our lives. Speculating, for instance, on which two commandments would have been omitted had we only eight fingers is enough to keep one awake at night.

A living example of this arbitrary nature of number bases was recently brought to light by the discovery of a long lost tribe living in the remote jungles of South America. It would seem the tribe had been isolated from the rest of the world for at least 10,000 years. An interesting aspect of their life was a huge population of dogs living among the people. In fact, dogs so outnumbered the people (so to speak) that the people had evolved a counting system based on the number of legs on a dog, as opposed to our more rational base ten. They counted in the equivalent of base four.

In counting, they would be heard to say, "one, two, three..." Since they had never developed more than four symbols to count with (0,1,2,3) when they got to the number after three, they wrote it as 10 and called it *doggy*, thus confirming the quantity in terms of a natural unit in their environment. Continuing to count they would say, "doggy-one (11), doggy-two (12), doggy-three (13)..."

At this point they would write the next number as 20 and call it *twoggy*. A similar procedure was used for 30.

20—twoggy	30—troggy
21—twoggy-one	31—troggy-one
22—twoggy-two	32—troggy-two
23—twoggy-three	33—troggy-three

Now, upon reaching 33, the next number must again force another position in the number display.

You're probably wondering what they called it. The digits are of course 100. Oh, the name? Why, of course, it's *one houndred*.



CHAPTER 2

Assemblers

I mentioned earlier that the basic principle of the Apple is its ability to scan through a range of memory and execute different operations depending on what numeric values it finds at each location, or *address*. Instead of tediously loading each location by hand with mundane numbers to create a program, an assembler is used to translate abbreviated codewords, called *mnemonics*, into the proper number values to be stored in memory.

The types of assemblers available are quite diverse, and range from the Mini-Assembler present in an Apple with Integer Basic (or the *Munch-A-Bug* package) to sophisticated editor/assemblers like *Merlin*.

For now, we'll use the Mini-Assembler to try a short program. For writing very simple machine language programs, Integer Basic Apples have a built-in Mini-Assembler. If all you have is Applesoft, this is not available. In either case, you'll want to get a more complete assembler to do any real program writing.

Starting with chapter three, I'll assume you have an assembler, and have learned at least enough about operating it to enter a program. Since the only two commands we have at this point are JSR and RTS, our routine will be very simple. In the Monitor at \$FBDD is a routine that beeps the speaker. Our routine will do a JSR to that subroutine, then return to Basic via an RTS at the end.

To enter the program using the Mini-Assembler, follow these

steps:¹ From Integer, enter the Monitor with a CALL - 151. Then type in:

F666G

F666 is the address where the Mini-Assembler program starts. G tells the Monitor to execute the program there. You can think of G as go; its Basic equivalent is RUN. The general syntax is:

<start address>G

The prompt should change to an exclamation mark (!). To use the Mini-Assembler, you must follow a basic pattern of input. See page 49 in the newest *Apple II Reference Manual* for a thorough description of this. For now, though, enter:

```
!300: JSR FBDD <RETURN>
```

The Apple will immediately rewrite this as:

```
0300-- 20 DD FB JSR $FBDD
```

The input syntax is to enter the address at which to start the program followed by a colon and a space, then enter the mnemonic, another space, and then the operand, in this case the address for the JSR to jump to.

Next type in:

```
!RTS <RETURN>
```

which will be rewritten as:

```
0303-- 60 RTS
```

Be sure to enter one space before the RTS. What the assembler has done is to take our mnemonic input and translate it into the

1. If you do not have the Mini-Assembler available, you can enter the same data into memory by entering the Monitor and typing in:

```
300: 20 DD FB 60 <RETURN>
```

Rejoin us at the 300L mark on pg. 15

numeric opcodes and operands of actual machine language.

Now type in:

```
!$FF59G
```

This will exit the Mini-Assembler, giving you back the asterisk prompt (*) of the Monitor. You can now LIST your program by typing in:

```
300L
```

The first two lines of your listing should be:

```
0300- 20 DD FB JSR $FBDD
0303- 60 RTS
```

What follows after \$303 is more or less random and does not affect the code we have typed in. When run, this program will jump to the beep routine at \$FBDD. At the end of that routine is an RTS that will return us to our program at \$303. The RTS there will then do a final return from the program back to either the Monitor or Basic depending on where we call it from.

From the Monitor type in:

```
300G
```

The speaker should beep and you will get the asterisk prompt back. Now go back into Basic with a control-B. Type in:

```
CALL 768
```

The speaker should again beep and then give you the Basic prompt back. CALL 768 should work from Integer or Applesoft.

As long as the programs are not very involved, the Mini-Assembler is handy for writing quick routines. A complete table of routines in the Monitor appears in appendix D at the end of the book. Try to write your own JSR's to one or more of these routines. You might even try doing several in a row for fun.

Now let's look at the operation of a more typical assembler. This example assumes you're using an assembler similar to the ones mentioned in the introduction. If you have a different assembler that gives you different results, you may have to con-

sult your operating manual for the proper procedures for entering source listings.

Before presenting the listing, I'd like to clarify two commonly used terms in assembly language programming, *source code* and *object code*. Source code is the English-like text you enter into the assembler. This text has the advantage of being easily readable, and may include whole sentences or paragraphs of comments very similar to REM-type statements found in Basic. Source code is, however, not directly executable by the 6502. It simply does not understand English-like text. As mentioned earlier, the 6502's preferred (and in fact only acceptable) diet is one to three byte chunks of memory in which simple and unambiguous numbers are found.

The assembler takes this text and produces the pure numeric data, called the object code, which is directly executable by the 6502.

Now the listing:

Object Code	Source Code
	1 *****
	2 * SAMPLE PROGRAM *
	3 *****
	4 *
	5 *
	6 OBJ EQU \$300
	7 ORG EQU \$300
	8 BELL EQU \$FBDD
	9 *
300— 20 DD FB	10 START JSR BELL ; RING BELL
303— 60	11 END RTS ; RETURN

To the right side of the listing is what is generally called the source code. This is the program, coded using mnemonics and various names or labels for different parts of your routine. Very few actual addresses or values are used in the source code.

To the left is the object code. This is what is actually put in memory as the machine language program. The object code is what the computer actually executes; it is obviously rather difficult to understand, at least compared to trying to understand it when you have the advantage of the source code. Being more readily able to understand the coding places greater importance

on having the source listing for a given program and explains why your *Apple II Reference Manual* contains a source listing for the Apple Monitor. Such listings were considered necessary in documenting a system when the Apple came out.

However, source listings for Applesoft, Integer, and the Disk Operating System (DOS) are much harder to come by and are generally considered private property by Apple.

Most assemblers display both the object code and the source code when the ASM (for assemble) command is used. Object and source code are, however, usually saved to disk as two separate and distinct files. Initially, let's consider just the source listing.

The first thing to notice is that, just like in Basic, we again have line numbers. In assembly language, though, the line numbers are solely for use with the program editor, and are not used at all to reference routines. Inserting a line is done with a special editor command, and all following lines are automatically renumbered to accommodate the new line.

Next notice the syntax, or proper ordering of the information. Generally the syntax consists of three basic elements, or *fields*, to each line. These fields are either defined by their position on the line or, more often, by *delimiters*. A delimiter is a character used to separate one field from another. In most assemblers, a space is used. Using this convention, you don't have to tab over to some specific position for each field on the line. Instead you just make sure each field is separated from the adjacent one by a space.

The first field is for a *label* and is optional. Lines 10 and 11, for example, each have a label that applies to that point in the routine. In this case, the label START indicates where we first begin the program. END is the clever label used for the finish. You may even recognize this program as the one we used to beep the speaker earlier. Some assemblers limit the number of characters used in the label.

As the program becomes more complex, we can do the equivalents of GOTO and GOSUB by using these labels instead of a line number. You'll notice that to do this, BELL has to be defined somewhere in the listing. Since BELL does not occur as a label within our own program (lines 10 and 11), it is defined at the beginning using the EQU (EQUALS) statement. The statement reads: "BELL EQUALS \$FBDD." This way, whenever we use the label BELL, the assembler will automatically set up the JSR or whatever to the address \$FBDD.

The second field is the *command field* which includes the opcode and its operand. In line 10, the JSR is the opcode and the operand is BELL. Not all opcodes will have an operand.

The third field, to the right, is the *comment field*. Use of the comment field is optional and is reserved for any comments about the listing you might wish to make (for example, RING BELL). The semicolon in the source code is used as the delimiter for the comments field. Comments can also be done at the very beginning of the line by using an asterisk as the REMARK character. As in Basic, everything after the asterisk is ignored by the assembler.

Assemblers also have what are sometimes called *pseudo opcodes*, like EQU. Although pseudo opcodes do not translate into 6502 code, they are interpreted by the assembler according to assigned definitions as the object code is assembled.

Another use of pseudo opcodes involves OBJ and ORG on lines 6 and 7 of the source listing. OBJ stands for OBJECT and defines where the object code will be assembled in memory. In this case, the code will be assembled starting at the address \$300. ORG stands for ORIGIN and defines the base address to be used when creating the JSR's, JMP's, and other functions that reference specific addresses within the program. Generally OBJ and ORG are the same, and for the time being we'll leave it at that. Consult your assembler manual for more specific information on use of these commands.

Remember, only the actual program is converted into the object code. The remarks and the EQU, OBJ, and ORG statements are only used in the source code and are never transferred to the object code.

Load/Store Opcodes

One of the most fundamental operations in machine code is transferring the number values between different locations within the computer. You'll recall that in addition to the 64K of actual memory locations, there were registers inside the 6502 itself. These were the Accumulator, the X-Register, and the Y-Register. There are a number of opcodes that will load each of these registers with a particular value, and, of course, another set to store these values somewhere in the computer. The table below summarizes these:

	Accumulator	X-Register	Y-Register
to load:	LDA	LDX	LDY
to store:	STA	STX	STY

The first mnemonic, LDA, stands for Load Accumulator. LDA is used whenever you wish to put a value into the Accumulator. Conversely, to store that value somewhere, you would execute the STA command, which stands for Store Accumulator. The opcodes for the X-Register and Y-Register are similar and perform the identical function with the associated registers.

Now the question is, how do we control what numbers get put into the register we're concerned with? There are basically two options. The first is to put a specific number there. This is usually indicated in the source listing by preceding the number we want to be loaded with a "#" character.

```

99     LDA #$05      ;LOAD ACC. WITH THE
100    ;VALUE '05'
```

For instance, in this example, we have loaded the Accumulator with the value 5. How do you think we would load the X-Register or the Y-Register with the value 0?

The other alternative is to load the register with the contents of another memory location. To do this, we just leave off the "#" character.

```

99     LDA $05      ;LOAD ACC. WITH THE
100    ;CONTENTS OF LOC. $05
```

In this case, we are loading the Accumulator with whatever location \$05 happens to be holding at the moment.

These two options are called *addressing modes*. The first example (#\$05) we call the *immediate mode*, because it is not necessary to go to a memory location to get the desired value. The second case we call the *absolute mode*. In this mode, we put a given value in the register by first going to a specified memory location that holds the value we want.

Putting It All Together

We now have the ability to transfer numbers about in the computer, to jump to other subroutines within the Apple via a

JSR, and to return safely to the normal world via an RTS when we're done. In addition, we have an assembler that will allow us easily to generate a source listing for our program, which can also be easily modified. Let's put all this together to write a short program to print some characters on the screen. Appendix E contains two charts (the ASCII Screen Character Set, and the Text Screen Map) that will supply the necessary information to achieve this.

When a character is printed on the screen, what is really happening is that a number value is being stored in the area of memory reserved for the screen display. Change a value there and a character on the screen will change. The Text Screen Map gives the various addresses of each position on the screen. The upper left corner corresponds to location \$400, the lower right to \$7F6.

The ASCII Screen Character Set shows which number values create which screen characters. Suppose we want to print the word APPLE in normal text. The chart indicates that we should use the following values:

```
A—$C1
P—$D0
P—$D0
L—$CC
E—$C5
```

If we want the word to appear on the seventh line of the screen, we should load these values into locations \$700 to \$704. To test this, enter the following program using your assembler. If you still don't have one, the Apple Mini-Assembler can be used, although we will soon reach the point where it will not be sufficient for our needs. If you are using the Apple Mini-Assembler, enter only the program itself, ignoring the OBJ, ORG comments. In place of JSR HOME enter JSR \$FC58.

At the beginning of the program, we define where it is to be assembled. Then we define a routine in the Apple called HOME, which is part of the Apple Monitor and is at \$FC58. Whenever this routine is called, the screen is cleared and the cursor put in the upper left corner. This ensures us that only the word APPLE will be printed on the screen.

```
1 *****
2 * TEST PROGR. #1 *
3 *****
```

```

                ORG     $300
                4 OBJ     $300
                5 HOME    EQU     $FC58
                6 *
300-   20 58 FC   7 START  JSR     HOME ; CLEAR SCREEN
303-   A9 C1    8        LDA    #$C1 ; 'A'
305-   8D 00 07 9        STA    $700
308-   A9 D0   10       LDA    #$D0 ; 'P'
30A-   8D 01 07 11      STA    $701
30D-   8D 02 07 12      STA    $702
310-   A9 CC   13       LDA    #$CC ; 'L'
312-   8D 03 07 14      STA    $703
315-   A9 C5   15       LDA    #$C5 ; 'E'
317-   8D 04 07 16      STA    $704
31A-   60      17 END    RTS

```

The routine will begin by doing a JSR to the HOME routine to clear the screen. Then the Accumulator will be loaded with an immediate \$C1, the value for the letter A. This will then be stored at location \$700 on the screen, which will cause the letter A to be visible on the screen. The next value loaded is for the letter P, and this is stored at \$701 and \$702. It is not necessary to reload the Accumulator, since storing the number does not actually remove it from the Accumulator. The number is just duplicated at the indicated spot. The process continues in this pattern until all five letters have been printed, and then an RTS returns us to normal operation.

Once you have assembled the routine at \$300, try calling it both from the Monitor level with:

```
300G
```

and from Basic (either one) with:

```
CALL 768
```

You should also change the LDA/STA to the X-Register and Y-Register equivalents to verify that they work in a similar manner.

Summary

You now have at your disposal a total of eight opcodes and a familiarity with assemblers. These few opcodes are probably the most often used, and, with just these alone, you can do quite a number of things. The JSR allows you to make use of all the

routines already available in the Monitor. I highly recommend getting *The Apple Monitor Peeled* by W.M. Dougherty, now available exclusively from Apple, for more information on using these routines. His book gives a lot of detail on what is available.

In chapter three, we'll look at some more advanced addressing techniques, and how to do counters and loops.



CHAPTER 3

Loops and Counters

Now we get into not only more mnemonics, but the techniques of using them to accomplish various overall operations. In particular, we'll look at counters and loops in assembly language. In Basic, the FOR-NEXT loop is one of the more essential parts of many programs, and this is no less true in machine programming. The only difference is how the loop/counter combination is actually carried out.

In Basic, the testing of counters is done either by IF-THEN statements or, automatically, in the NEXT statement of the FOR-NEXT loop. In assembly language, the testing is done by examining flags in the Status Register. These flags indicate the status of the various registers and memory locations. The Status Register is a fourth register of the 6502, one we have not previously mentioned. Before going on with loops and counters, it will be necessary to briefly discuss the Status Register, and in addition, binary numbers.

Like the other three registers—the Accumulator, the X-Register, and the Y-Register—the Status Register holds a single byte. You'll recall that each byte in the Apple can have a value from 0 to 255 (\$00 to \$FF).

As it happens, there are many ways of looking at and interpreting numbers. The one of common experience is that in which we consider only the magnitude of the number. Noticing that 255 is larger than 128 gives us only a very simple form of infor-

mation—whether a number is either less than, equal to, or greater than another number.

A second way of looking at numbers is in binary form. Base two allows us to see more information in a number and, hence, is that much more useful. We have already seen how a single byte can be represented either as 0 to 255 or as \$00 to \$FF. In binary the range is 00000000 to 11111111. For instance 133 (base ten) was represented as \$85. In binary it has the appearance 10000101. In this case, each 1 or 0 represents the presence or absence of a given condition. Thus, eight distinct pieces of information are conveyed, as well as all the various combinations possible.

Before you run shrieking from the room, remember that this is all done to make things easier, not harder. Besides, learning base sixteen (hex) wasn't that bad back at the beginning of this book, was it? So let's take a moment to see what this bits and bytes stuff is all about.

Binary Numbers.

The Apple is an electronic device and, actually, in many ways, a simple one at that. In most parts of its circuitry, the current is either off or on. That's it. No in-between. Having two possible positions is perfect for base two. The idea of a number base has to do with how many symbols, or units, you use for counting. We normally use ten. We have a total of ten possible symbols to write in a single position before we have to start doubling up and using two positions to represent a number. You'll recall in hex that, by using 0 through 9 and A through F, we had sixteen possibilities; thus, we were in base sixteen. With the on/off nature of the Apple, we're limited to two possibilities: 0 or 1.

How high can we count in one position? Not very. We start at 0, then go to 1, and that's it. Then we have to add another position. The next number, therefore, is 10. As before, remember that, in this case, 10 represents what we usually call two. If we use three positions, the lowest number is 100 (representing the quantity four in base ten).

For a given number base, there is a formula for the highest decimal number you can represent with a given number of positions.

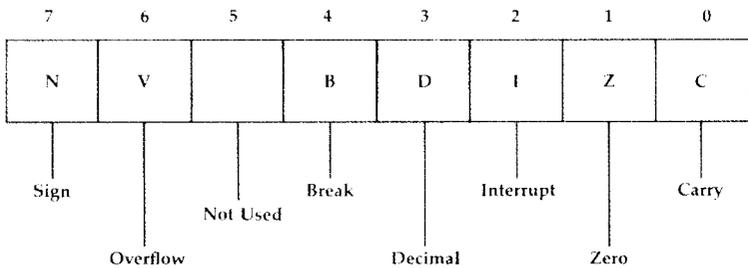
$$N = B^p - 1$$

... where N is the largest decimal number, B is the number base, and p is the number of positions available.

By using eight positions, we can go up to 11111111, which just happens to equal 255. How handy! This is the same maximum value as our bytes. And, if the truth be known, it's actually the other way around. We use the numbers 0 through 255 because we are using eight bits to make up each byte. Whether each bit is a 0 or a 1 depends on whether the part of the circuit that is responsible for that bit is off or on.

The Status Register.

Here at last is our representation of a single byte, made up of eight bits. In particular, the byte we are looking at is the Status Register of the 6502. The important difference between this register and the others is that it is not used to store number values. Instead it indicates various conditions.



The bits of the Status Register are numbered from right to left, 0-7. Each bit in this register indicates the status and/or results of different operations and is called a flag. It is by using this register that we can create counters and loops in our programs. The flag we will be immediately concerned with is bit one, the *zero flag*. In terms of the commands we already know, the zero flag is affected by an LDA, LDX, or LDY.

If the value loaded into the Accumulator, X-Register, or Y-Register were \$00, the flag would be set to one. If it were a nonzero number, the flag would be zero. Seemingly backward perhaps, but remember, each flag is set to show the presence or absence of a given condition, in this case, \$00. The setting or clearing of each Status Register flag is done automatically by the 6502 after each program step, indicating the results of any particular operation.

Incrementing and Decrementing.

To create a *counter* and then a *loop*, we will use the Status Register to tell when a given register or memory location reaches zero. We will also need a way of changing the value of the counter in a regular fashion. In the 6502, this is done by *incrementing* or *decrementing* by one each time, as indicated.

	Accumulator	Y-Register	X-Register	Memory Location
Increment by one:	Not Available	INX	INX	INC
Decrement by one:	Not Available	DEX	DEY	DEC

This table shows the mnemonics used to increment or decrement a particular register or memory location.

Note that directly incrementing or decrementing the Accumulator is not possible. The increment/decrement commands affect the zero flag, depending on whether the result of the operation is zero or not.

The usual syntax for using these commands in an assembly listing is:

```
10 INX
11 INY
12 DEX
13 DEY
14 INC $0600
15 DEC $AA53
```

For the register operations, the command stands alone, with no need of an operand. In the case of *INC* and *DEC*, the memory locations to be operated on are given, in hex of course, usually preceded by the dollar sign.

One thing to mention here is the *wrap-around* nature of the operations. To understand this, examine the following chart:

Original Contents	Increment	Decrement	Z-Flag set?	Z
\$05	\$06	\$04	no, no	0,0
\$0F	\$10	\$0E	no, no	0,0
\$01	\$02	\$00	no, yes	0,1
\$FF	\$00	\$FE	yes, no	1,0
\$00	\$01	\$FF	no, no	0,0

The effects of incrementing and decrementing different values are shown, along with the effects on the zero flag after the operations. The first case is simple, $5 + 1 = 6$, $5 - 1 = 4$. In both cases, the result is nonzero, so the zero flag is not set. For \$0F, the same holds true. Remember that, in hex, the next number after \$0F is \$10. In the case of \$01, incrementing produces \$02. When we decrement \$01, the result is \$00; the zero flag is set.

Here's where it gets interesting. When the starting value is \$FF, adding one would normally give \$100. However, since a single byte only has a range of \$00 to \$FF, the 1 is ignored, and the value becomes \$00. This sets the zero flag. In the case of decrementing, $\$FF - 1 = \FE , so the zero flag is not set.

If we start with \$00, although incrementing produces the expected \$01, decrementing wraps around in the reverse of the previous case, giving \$FF. Both results are nonzero, so Z—short for the zero flag—is clear, that is, not set, for both operations.

Looping with BNE

The only procedure remaining to enable you to create a loop is a way of testing the Z-flag and then being able to get back to the top of the loop for another pass. In Basic, a simple loop might look like this:

```
10 HOME
20 X = 255
30 PRINT X
40 X = X - 1
50 IF X <> 0 THEN GOTO 30
60 END
```

In this program, we start with the counter X set at 255. Then the value is printed, decremented, and the process repeated until the counter reaches zero. We can make the loop execute any number of times by properly setting the initial value of X.

In machine code, the test and GOTO is done with a branch instruction. In this case, the one we'll use first is BNE. BNE stands for Branch Not Equal and is a branch instruction executed when a register is loaded with the value zero. This can happen either directly with something like a "LDA #\$00" or as the result of an arithmetic operation, such as an INX, DEC, or ADC. Here is the assembly language equivalent of the Basic listing:

```

1 *****
2 * LOOP PROG. 1 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 HOME EQU $FC58
8 *
9 START JSR HOME
10  LDX #$FF
11  LOOP STX $700
12   DEX
13   BNE LOOP
14  END RTS

```

And here is the way Apple's disassembler would show it:

```

*300L
0300- 20 58 FC   JSR   $FC58
0303- A2 FF     LDX   #$FF
0305- 8E 00 07  STX   $0700
0308- CA       DEX
0309- D0 FA     BNE   $0305
030B- 60       RTS

```

In this program, we first do a JSR to the clear screen routine in the Monitor that we used in chapter two. Then we load the X-Register with a starting value of \$FF. Now we start the loop. Storing the X-Register at \$700 will make the loop's action visible as a character on the screen for each pass through the loop. Next, DEX subtracts one from the current value of the X-Register. The BNE will then continue the loop back up to LOOP until the X-Register reaches \$00, at which point the test will fail, and program execution will fall through to the RTS at the end of the program. People will also refer to the execution of a branch instruction saying the branch is ignored or taken depending on whether program flow falls through the branch instruction, or goes to the new address indicated by the branch instruction.

Try entering this now, and also notice how fast the program runs. You probably weren't able to see very much, but all 255 values were put to the screen. The inverse A that's left on the screen is how a \$01 at \$700 appears. (\$00 doesn't get printed—why?) To verify that each pass is being executed, replace the STX \$700 in the source listing with a JSR \$FBDD. If you don't want

to hear 255 beeps, try changing the initial value of the X-Register in line 10. As before, you should be able to call this program from the Monitor with a 300G, or from Basic with a CALL 768.

You may also wish to try the equivalent version of the program, using the Y-Register or a memory location as the counter. I would suggest trying to write a program using INC, INX, or INY to drive the counter as a practice program.



CHAPTER 4

Looping with BEQ

In chapter three, we started into the various techniques of creating and using counters and loops in machine language. To accomplish the loop, we used the value in one of the registers as a counter and the branch instruction that tests for the presence of a nonzero number in the register to actually do the looping. Recall that this evaluation of zero/nonzero is done via the zero bit, or flag, of the Status Register of the 6502.

The complement of the BNE instruction is something called BEQ, which obscurely enough stands for Branch Equal. It operates in just the opposite fashion from BNE; that is, it branches only when the register or memory location reaches a value of zero.

For example, consider this Basic listing:

```
10 HOME
20 X = 255
30 PRINT X
40 X = X - 1
50 IF X = 0 THEN 70
60 GOTO 30
70 END
```

In this case, the loop continues as long as X is not equal to zero. If it is, the branch instruction is carried out and the program ends. In assembly language, this program would be the equivalent:

```

1 *****
2 *LOOP PROG. 2*
3 *****
4 *
5   OBJ $300
6   ORG $300
7   HOME EQU $FC58
8 *
9   START JSR HOME
10  LDX #$FF
11  LOOP STX $700
12  DEX
13  BEQ END
14  JMP LOOP
15  END RTS

```

Notice that this program requires the addition of a new instruction to our repertoire, the `JMP` command. This is analogous to a `GOTO` in Basic, and in this program will cause program execution to jump to the routine starting at `LOOP` each time. Only when the X-Register reaches zero does the `BEQ` take effect and cause the program to skip to the `RTS` at end. Here is the way this would appear when put into memory, and then listed with the `L` command from the Monitor:

```
*300L
```

```

0300- 20 58 FC   JSR      $FC58
0303- A2 FF     LDX      #$FF
0305- 8E 00 07   STX      $0700
0308- CA       DEX
0309- F0 03     BEQ      $030E
030B- 4C 05 03   JMP      $0305
030E- 60       RTS

```

The assembler automatically translates the positions of `LOOP` and `END` into the appropriate addresses to be used by the `BEQ` and `JMP` when it assembles the code.

Remember that to the left are the addresses and the values for each opcode and its accompanying operand. The more intelligible translation to the right is Apple's interpretation of this data.

Branch Offsets and Reverse Branches

Notice that the `JMPs` and `JSRs` are immediately followed by the

addresses (reversed) that they are to jump to, such as in the first JSR as \$300.

However, branch instructions are handled a little differently. The \$03 is an offset that tells the 6502 to jump three bytes past the next instruction.

Since the next instruction is at \$30B, the 6502 will branch to \$30E, thus skipping the JMP command and going directly to the RTS, which terminates the routine.

Branches can also be done in the reverse direction. Here is a rather inefficient, but illustrative example:

```
1 *****
2 * LOOP PROG. 2A *
3 *****
4 *
5   OBJ $300
6   ORG $300
7   HOME EQU $FC58
8 *
9   START JSR HOME
10      JMP SETX
11   END RTS
12 *
13   SETX LDX #$FF
14   LOOP STX $700
15   DEX
16   BEQ END
17   JMP LOOP
```

The Monitor listing for this would be:

```
0300- 4C 04 03   JMP      $0304
0303- 60          RTS
0304- A2 FF      LDX      #$FF
0306- 8E 00 07   STX      $0700
0309- CA          DEX
030A- F0 F7      BEQ      $0303
030C- 4C 04 03   JMP      $0304
```

In this example, the branch, if taken, will cause the program to move back up through the listing. To indicate this branch in the opposite direction, the high bit is set. This is the same technique that is often used to show negative numbers in machine language programs. Please note that it is not just a matter of setting the high bit. If that were the case, the value following the BEQ command might be expected to be \$89. (The address of

the next instruction (\$30C) minus where we want to go to (\$303) equals \$09. Then with the high bit on, we have \$89.)

This is almost correct. The actual value is arrived at by subtracting the branch distance from \$100. Thus \$100 minus \$09 equals \$F7. This is so that the destination address can still be arrived at through addition. Notice that \$30C + \$F7 = \$403. It is then very easy for the 6502 to correct this back one page to \$303.

If all this seems a bit confusing, try not to let it bother you. In actual practice, there is not much reason to be concerned about the way in which the offset byte is determined since your assembler will determine the proper values for you when assembling code, and Apple's disassembler, as well as many others, including *Sourceror*, will give the destination address when reading other code.

This is also a good time to stress the importance of working through each of these examples on your own, step by step, to make sure you understand exactly what happens at each step, and how it relates to the rest of the program. If you're not sure, go back over it until that proverbial light comes on!

Screen Output Using COUT

As the X-Register is incremented in this program, we'll stuff the value into \$700 so we can see something on the screen as the counter advances.

Now you may remark from your experience in chapter three, that although this program is pleasantly simple in its logic, it is not much fun to watch on the screen because it runs so quickly.

To solve this, we will start to make more extensive use of the routines already present in the Monitor to do certain tasks, and thus make our programming requirements a little simpler. Referring to the Monitor subroutines in appendix D, it happens that the first routine listed is something called COUT. This is the routine that actually sends a character we want output to whatever device(s) may currently be in use. Most of the time, this just goes directly to the next routine listed, COUT1 (clever with the names, aren't they?), which specifically handles the screen output. What this means for us is that anytime we want to output a character, we don't have to write our own routines to worry about all the in-depth details about the screen (cursor position, screen size, whether it's time to scroll), we just load the Accu-

mulator with the ASCII value for the character we want to print, and then do a JSR \$FDED!

Now comes some programming technique. We would like to have the counter value in the Accumulator so we can print it via COUT, but unfortunately our increment/decrement commands only work for the X-Register, the Y-Register, and given memory locations. To solve this, we'll have to expand our listing a little. This time, we'll use a memory location as the counter, and then load the Accumulator, on each pass through, to print out a visible sign of the counter's activity. Good locations to use for experimenting are \$06 to \$09. These are not used by either Integer, Applesoft, DOS, or the Monitor. This is important for avoiding conflicts with the Apple's normal activities while running your own programs.

And now our revised listing:

```
1 *****
2 *LOOP PROG. 2B*
3 *****
4 *
5   OBJ $300
6   ORG $300
7   CTR EQU $06
8   HOME EQU $FC58
9   COUT EQU $FDED
10  *
11  START JSR HOME
12   LDA #$FF
13   STA CTR
14  LOOP LDA CTR
15   JSR COUT
16   DEC CTR
17   BEQ END
18   JMP LOOP
19  END RTS
```

Apple's L command will give this after you've assembled it in memory:

```
*300L

0300-- 20 58 FC   JSR   $FC58
0303-- A9 FF     LDA   $FF
0305-- 85 06     STA   $06
0307-- A5 06     LDA   $06
```

0309-	20	ED	FD	JSR	\$FDED
030C-	C6	06		DEC	\$06
030E-	F0	03		BEQ	\$0313
0310-	4C	07	03	JMP	\$0307
0313-	60			RTS	

A call to this routine via our usual 300G from the Monitor, or a CALL 768 from Basic should clear the screen, then print all the available characters on your Apple, in all three display modes (normal, flashing, and inverse). The beep you hear is the control-G (bell) being *printed* to the screen via COUT. The invisible control characters account for the blank region between the two main segments of output characters. You will also see some characters that are not normally generated by the Apple, such as underscore, reverse slash, and the left square bracket (____, \ , []).

The alphabet is backward because we started at the highest value and worked our way down. From chapter three, though, you'll remember that when a byte is incremented by one from \$FF, the result *wraps around* back to \$00. This will produce an action testable by a BEQ. Using this wrap-around effect of the increment command, we can rewrite the program to be a little more conventional like so:

```

1 *****
2 *LOOP PROG. 3*
3 *****
4 *
5   OBJ $300
6   ORG $300
7   CTR EQU $06
8   HOME EQU $FC58
9   COUT EQU $FDED
10 *
11 START JSR HOME
12   LDA #$00
13   STA CTR
14 LOOP LDA CTR
15   JSR COUT
16   INC CTR
17   BEQ END
18   JMP LOOP
19 END RTS

```

With the Apple showing:

*300L

```
0300- 20 58 FC JSR $FC58
0303- A9 00 LDA #$00
0305- 85 06 STA $06
0307- A5 06 LDA $06
0309- 20 ED FD JSR $FDED
030C- E6 06 INC $06
030E- F0 03 BEQ $0313
0310- 4C 07 03 JMP $0307
0313- 60 RTS
```

A call to this routine should now print out the characters in a more familiar manner. At last our programs are starting to do something interesting! It gets better!

Reading a Game Paddle

Let's try reading a game paddle, and use what we get back to print something to the screen! Granted, I'm not any more sure than you are what good this might be, but it's guaranteed to be a new program in your library!

The PREAD subroutine in appendix D indicates that a paddle can be read by loading the X-Register with the value for the paddle you wish to read, followed by a JSR \$FB1E. When the routine returns, the value of the paddle will be in the Y-Register. All we have to do then is grab this, stuff in the Accumulator, and then do our JSR COUT.

```
1 *****
2 *PADDLE PROG. 1*
3 *****
4 *
5 OBJ $300
6 ORG $300
7 TEMP EQU $06
8 PREAD EQU $FB1E
9 HOME EQU $FC58
10 COUT EQU $FDED
11 *
12 START JSR HOME
13 LDX #$00
14 LOOP JSR PREAD
15 STY TEMP
16 LDA TEMP
17 JSR COUT
18 JMP LOOP
19 *INFINITE LOOP
```

You should get this in memory:

*300L

0300-	20	58	FC	JSR	\$FC58
0303-	A2	00		LDX	#\$00
0305-	20	1E	FB	JSR	\$FB1E
0308-	84	06		STY	\$06
030A-	A5	06		LDA	\$06
030C-	20	ED	FD	JSR	\$FDED
030F-	4C	05	03	JMP	\$0305

This routine when called will quickly fill up the screen and then change the stream of characters output as you turn paddle zero. Since we have no test for an end, reset is the only way to stop this infinite loop.

Depending on your propensity toward being hypnotized, you may lose touch with the world for indefinite periods of time while running this program. At the inverse and flashing end, it's also remarkably good at stimulating migraine headaches in record time. By carefully controlling the paddle, you can also observe some interesting bits of ASCII trivia. For example, after the inverse and flashing range, you should be able to stop the flow by moving into the control character range. With sufficient dexterity, you should be able to lock onto the persistent beep of the bell (control-G).

Shortly after this point, the screen will zip into motion when you hit the line feed character (control-J) and, of course, also at control-M (carriage return). What fun, eh! When normal character output returns as you pass the halfway point, you can delight in various patterns of screen filling. Why, you may even wish to try writing your name by deft control of the paddle—child's play!

Paddle Program Problems

Returning to reality here, it is worth mentioning that some problems in accuracy can arise from repeatedly reading the paddle so quickly. The analog circuits don't have time to return to zero, and various problems creep in.

Also, we have been a bit negligent in looking out for conflicting use of the registers by the various routines we are calling. There is often no assurance that the register you're using for your own routine won't be clobbered by the Monitor routine you

use. In the case of the paddle and output routines, you'll note they did mention how the X-Register, the Y-Register, and the Accumulator were affected by each of the routines.

For the record, here is a reasonable facsimile of our program in Applesoft:

```
10 HOME
20 T = PDL(0)
30 PRINT CHR$(T)
40 GOTO 20 ;
```

It is also worth mentioning that our machine language version takes eighteen bytes, while the Applesoft one takes thirty-eight, not counting space used by the variable T.

Execution speed may seem to be similar, but this is because of the printing of the characters to the screen. In most cases, machine execution would be orders of magnitude faster.

Transfer Commands

In our program, we have to go through a rather inelegant way of transferring the value from the Y-Register to the Accumulator, using a temporary storage byte. Fortunately, there is an easier way. There are four commands for transferring contents of the X-Register or the Y-Register to and from the Accumulator. They are as follows:

```
TXA: Transfers contents of X-Register to Accumulator.
TYA: Transfers contents of Y-Register to Accumulator.
TAX: Transfers contents of Accumulator to X-Register.
TAY: Transfers contents of Accumulator to Y-Register.
```

Each of these actions conditions the zero flag upon execution, so it is possible to test what has been transferred. There is no command to transfer directly between the X-Register and the Y-Register.

This gives us an even shorter program:

```
1 *****
2 * PADDLE PROG. 1A*
3 *****
4 *
5 OBJ $300
```

```

6   ORG $300
7   PREAD EQU $FB1E
8   HOME EQU $FC58
9   COUT EQU $FDED
10  *
11  START JSR HOME
12   LDX #$00
13  LOOP JSR PREAD
14   TYA
15   JSR COUT
16   JMP LOOP
17  * INE LOOP

```

Now it's only fifteen bytes long!

*300L

```

0300- 20 58 FC   JSR   $FC58
0303- A2 00     LDX  #$00
0305- 20 1E FB   JSR  $FB1E
0308- 98        TYA
0309- 20 ED FD   JSR  $FDED
030C- 4C 05 03   JMP  $0305

```

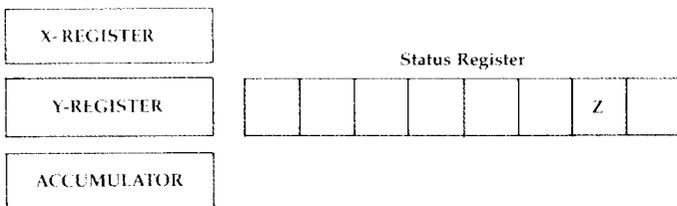
With twenty commands at your disposal, you now know just over a third of the total vocabulary of the language. Soon, you'll be dangerous!



CHAPTER 5

Comparisons; Reading the Keyboard

Now we're getting to where we can actually do some interesting things with what we know so far. The basic ideas you should be comfortable with at this point are fairly simple. The 6502 microprocessor is our main operational unit. There are three main registers: the Accumulator, the X-Register and the Y-Register. Also present is the Status Register, which holds a number of one-bit flags to indicate various conditions. So far, the only one we've considered is the Z-flag, for indicating whether a zero or nonzero number is present in one of the other three registers.



6502 Model

Programs are executed by the 6502 scanning through memory. Addresses in memory are analogous to line numbers in Basic. A JSR \$FC58 in machine language is just as valid as a GOSUB

1000 in Basic. In using an assembler, we can give names to routines at given addresses and make things that much simpler by saying JSR HOME, when HOME has been defined as \$FC58.

In chapter four, we used testing commands like BEQ and BNE to create simple loops. We used the X-Register and the Y-Register as counters, and incremented or decremented by one for each cycle of the loop.

Now let's expand our repertoire of commands by adding some new ones and, in the process, add some flexibility to what we can do with loops and tests in general.

In our previous programs, we relied on our counters reaching zero and testing via the Z-flag to take appropriate action. Suppose, however, that we wish to test for a value other than zero. This is done using two new ideas.

Compare Commands and Carry Flag

The first is the compare command, the mnemonic for which is CMP. This tells the computer to compare the contents of the Accumulator against some other value. The other value can be specified in a variety of ways. A simple test against a specific value would look like this:

```
CMP #$A0
```

This would be read, "Compare Accumulator with an immediate A0." This would tell the 6502 to compare the Accumulator to the specific value \$A0. On the other hand, we may want to compare the Accumulator with the contents of given memory location. This would be indicated by:

```
CMP $A0
```

In this case, the 6502 would go to location \$A0, see what was there, and compare that to the Accumulator. It is important to understand that the contents of \$A0 may be anything from \$00 to \$FE, and it is against this value that the Accumulator will be compared. In each case, the 6502 does the comparison by internally subtracting the specified value from the Accumulator.

The second important idea is that of the *carry flag*. The carry flag enables us to determine the result of the comparison. Right next to the Z-flag in the Status Register is the bit called the *carry*.



The carry is used during addition and subtraction by the 6502. In our case, since the compare operation involves subtraction, the carry flag can be used to test the result. You do this with two new branch commands, BCC and BCS. BCC stands for Branch Carry Clear. If the Accumulator is less than the value compared against, BCC will branch appropriately. BCS stands for Branch Carry Set and is taken whenever the Accumulator is equal to or greater than the value used.

This means that we can now not only test for specific values but also test for ranges. Try this example.

```

1 *****
2 * PADDLE PROG. 2A *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PREAD EQU $FB1E
9   HOME EQU $FC58
10  COUT EQU $FDED
11 *
12  START JSR HOME
13   LDX #$00
14  LOOP JSR PREAD
15   TYA
16   CMP #$C1;  CMP TO ASCII VAL FOR "A"
17   BCC LOOP;  TRY AGAIN IF LESS THAN
18   CMP #$DB;  CMP TO ASCII VAL FOR "I" ("Z" + 1)
19   BCS LOOP
20   JSR COUT
21   JMP LOOP
22 * INE. LOOP

```

When assembled and listed from memory, it should look like this:

```

*300L
0300-- 20 58 FC   JSR     $FC58
0303-- A2 00     LDX     #$00
0305-- 20 1E FB   JSR     $FB1E
0308-- 98         TYA
0309-- C9 C1     CMP     #$C1
030B-- 90 F8     BCC     $0305

```

030D-	C9 DB	CMP	#\$DB
030F-	B0 F4	BCS	\$0305
0311-	20 ED FD	JSR	\$FDED
0314-	4C 05 03	JMP	\$0305

Let's step through the program. After the JSR to the clear screen routine, we load X with zero in preparation for reading a paddle. The #\$00 will tell the routine that we wish to read paddle zero. After the read, the answer is returned in the Y-Register, which we transfer to the Accumulator with a TYA. It is at this point that we use our test section. If the Accumulator is less than the ASCII¹ value for the letter A, we avoid the printout by going back to LOOP. I have used the ASCII value for A plus \$80 so that we get normal output on the screen. If we test for \$41 instead, flashing characters will be output to the screen.

The next comparison is for the ASCII value for the character "[". This comparison assures that the BCS will catch all values higher than the one for Z. The first chart in appendix E (Keys and Their Associated ASCII Codes) is useful in seeing where these numbers come from.

Only numbers from \$C1 to \$DA will make it through to be printed out using COUT (\$FDED).

Again the loop is infinite, so reset is required to exit.

The X-Register and Y-Register can also be compared in a

1. ASCII (for American Standard Code for Information Interchange) is a coding scheme for transmitting text. It is also used in the Apple for encoding text in memory, screen display, disk files, printer output, and many other areas. Appendix E gives a chart of all the characters and their ASCII values. One important note. It is possible to encode all the alphabetic characters (upper and lower case), numerics, special symbols, and control codes using only 128 characters. This means that ASCII is considered a *7 bit code*. This means that all the information required to determine which character has been sent is contained in bits 0-6 of the byte. Thus \$8A is reasonably equivalent to \$0A as far as its ASCII interpretation is concerned. The matter of the high bit being set or clear can create considerable confusion when it is not made clear what the computer expects.

Generally the Apple operates internally with the high bit set on all characters. That is to say characters retrieved from the keyboard via \$C000 and characters stored in the screen area of memory and on disk all usually have the high bit set (i.e. value equal to or greater than \$80). This is also the way Applesoft stores data within program lines. To keep you on your toes though, Apple printer cards usually do not support the high bit set when sending output to a printer, and strings within a program (such as A\$ = CAT) also have the high bit clear. Also, when using COUT (the Monitor text output routine), the high bit should be clear (always load the Accumulator with values less than \$80 before calling COUT).

I wish I could say it was all easier than that, but then again if it were all that easy, you wouldn't have to have bought this book, and then where would I be?

similar manner by codes CPX and CPY. Can you rewrite this program to use CPY instead of CMP?

BEQ and BNE are also still usable after a compare operation. Here's a summary:

Command	Action
CMP	Compares Accumulator to something
CPX	Compares X-Register
CPY	Compares Y-Register
BCC	Branch if register < value
BEQ	Branch if register = value
BNE	Branch if register <> value
BCS	Branch if register >= value

Using Monitor Programs for I/O Routines

As you may have noticed, I enjoy using the paddles as input devices. This is because they're an easy way of sending values from \$00 to \$FF into the system in a very smooth and natural way. We can get similar data from the keyboard, though. There the advantage is that we can jump from one value to another, with no transition between the two values.

A good part of many formal machine language courses deals with system I/O—that is, getting data in and out via different devices. Writing such things as printer drivers, disk or tape access routines, hardware interface software, etc., are the areas that hardcore programmers spend their youths mastering. Using the Monitor routines on the Apple simplifies this for us greatly because we don't have to do a lot of I/O details. You've already shown this by using the paddles (\$FB1E) for input and the screen (\$FDED) for output without having to know anything about how the actual operation is carried out. The keyboard is even easier.

I mentioned earlier that the address range from \$C000 to \$FFFF is devoted to hardware—these memory ranges cannot be altered by running programs. (I'm ignoring the RAM cards for the time being.) The range from \$D000 to \$FFFF is used by ROM routines that we've been calling. The range from \$C000 to \$CFFF is assigned to I/O devices. Typically the second digit (or maybe I should call it a hexit) from the left gives us the slot number of the device. For instance, if you have a printer in slot one, a look at \$C100 will reveal the machine language code in ROM of the card that makes it work. At \$C600 you'll probably find the code that makes the disk drive in slot six boot.

\$C000 to \$C0FF is reserved not for slot zero, but for doing special things with the hardware portions of the Apple itself. An attempt to disassemble from \$C000 will not produce a recognizable listing, but it will probably cause your Apple to act a bit odd. This range is made up of a number of memory locations actually wired to physical parts of your Apple. If you type in:

```
* C030
```

from the Monitor, in addition to getting some random value displayed, the speaker should click. If it doesn't click the first time, try again. Each time you access \$C030, the speaker will click as it moves in response to your action.

The keyboard is also tied into a specific location. By looking at the contents of \$C000, you can tell if a key has been pressed. In Basic, it's done with a PEEK - 16384. (See page 6 of the 1981 *Apple II Reference Manual*.) In machine language, you would usually load a register with the contents of \$C000, such as:

```
LDA $C000
```

Reading Data from Keyboard

Because it is difficult to read the keyboard at exactly the instant someone has pressed the key, the keyboard is designed to hold the last key pressed until either another key is pressed or until you clear the *strobe*, as it's called, by accessing an alternate memory location, \$C010. The strobe is wired to clear any characters off the keyboard that may be hanging around for any number of various reasons. When you check for a character, you don't want to pick one up that someone inadvertently entered prior to your enquiry (perhaps by nervously drumming their fingers across the keyboard while waiting for one of Apple's lightning-like disk accesses!). It is also always a good idea to clear the keyboard when you're done with it, otherwise you may similarly have the key pressed for your input still hanging around for whatever reads the keyboard next, such as an INPUT statement in Basic. The strobe is cleared by *either* a read or a write operation. It is the mere access to \$C010 in any manner that accomplishes the clear. Thus a LDA \$C010 would work just as well as a STA \$C010.² The last point to be aware of is that the keyboard is set up to tell you when a key is pressed by the value

that is read at \$C000. Now, you might think that the logical way would be to keep \$A0 in \$C000. Perhaps, but that's not the way they do it. Instead, they add \$80 to whatever the ASCII value is of the key you pressed. If a value less than \$80 is at \$C000, it means a key has not been pressed.

So, to illustrate this (and I admit it got a little involved for my tastes), let's look at some sample programs to read data from the keyboard.

```
1 *****
2 * KEYBOARD PROG. 1A *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   KYBD EQU $C000
9   STROBE EQU $C010
10  COUT EQU $FDED
11  HOME EQU $FC58
12 *
13  START JSR HOME
14  LOOP LDA KYBD
15     CMP #$80
16     BCC LOOP
17     JSR COUT
18     JMP LOOP
19 * INE LOOP
```

Once entered, this should disassemble as:

```
*300L
0300-- 20 58 FC   JSR   $FC58
0303-- AD 00 C0   LDA   $C000
```

2. Having now just said that read and write operations are essentially equivalent for clearing the strobe, let me cover myself enough to say that there is one slight difference. A write operation actually accesses the location *twice*, whereas a read operation only accesses once. Most of the time this doesn't make any difference. Since most people can't type at 100,000 characters per second, it's hard to get a character in between the two clear operations. However, there are, now available for the Apple, *keyboard buffers* which will store a whole string of characters entered by the user, instead of the usual one normally used for the keyboard. As each character is read in, it is taken out of the buffer by clearing the strobe. You guessed it! A write operation—such as a `STA $C010` or a `POKE -16368,0`—will clear *two* characters out of the buffer: the one you just read *and* the next one in line. Therefore, it is generally good practice to clear the strobe with a read operation, such as a `LDA $C010,X=PEEK -16368`, or the like. Like I said, if it were too easy. . .

```

0306-   C9 80       CMP       #$80
0308-   90 F9       BCC       $0303
030A-   20 ED FD    JSR       $FDED
030D-   4C 03 03    JMP        $0303

```

Trying this program, you should notice that the program runs on, printing the same character until you press another key. That's because we never cleared that strobe you thought I was rambling on about. Once the key press gets on the board, it's never cleared until it is replaced by a new key.

A better program is:

```

1 *****
2 * KEYBOARD PROG. 1B *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   KYBD EQU $C000
9   STROBE EQU $C010
10  COUT EQU $FDED
11  HOME EQU $FC58
12 *
13  START JSR HOME
14  LOOP LDA KYBD
15    CMP #$80
16    BCC LOOP
17    STA STROBE
18    JSR COUT
19    JMP LOOP
20 * INE LOOP

```

which lists as:

```

*300L

0300-   20 58 FC    JSR       $FC58
0303-   AD 00 C0    LDA       $C000
0306-   C9 80       CMP       #$80
0308-   90 F9       BCC       $0303
030A-   AD 10 C0    STA       $C010
030D-   20 ED FD    JSR       $FDED
0310-   4C 03 03    JMP        $0303

```

This should work better. Here we clear the keyboard whenever we've gotten a character and printed it. Why not clear it right after the read on line 15? If we did that, we'd be lucky to catch

a glimpse of the character at \$C000 as the user pressed the key. As it is, we can probably get away with it because of the speed of the loop. But if we had to go away to another routine for a while, or otherwise delay getting back to the LDA \$C000, we'd probably miss it.

You should also type in enough to wrap around onto the next line, and also try the arrow keys and RETURN. You may think this all performs as expected (with the exception of the missing cursor), but this all should not be taken for granted. Without the screen management of COUT, you'd have to do quite a bit more programming to keep things straight. Once more, this is the advantage of using the routines already present in the Monitor, rather than worrying about the details yourself.

Also, please notice how the STA was chosen because we didn't want to lose the contents of the Accumulator in doing the access.

This information concerns technique more than actual commands, but is worth mentioning if you're going to get along with your Apple successfully.

On page 130 of the 1981 *Apple II Reference Manual* you'll find a listing of the soft switches and other goodies at \$C000-C0FF. These can be very useful in having your Apple relate to the outside world.

You may wish to experiment with these. Also don't forget about all the routines listed in appendix D. These are also fun to experiment with and are provided to encourage you to write short programs just to test your wings. As I've mentioned before, they're also useful in saving you the trouble of writing your own I/O and other more involved routines.

1000

900

800

700

600

500

400

300

200

100

0

100

200

300

400



CHAPTER 6

Addressing Modes

Let's look at the various addressing modes used in machine language programming. This concept is rather fundamental in programming and you may justifiably wonder why we have not covered it sooner. Well, as it happens, we have; I just didn't call it by name at the time. In chapter one we laid out the basic structure of sixty-four thousand individual memory locations. Since then, we've worked most of our magic by simply manipulating the contents of those locations.

Flexibility in the ways in which you can address these locations is the key to even greater power in your own programs.

Consider this chart of the addressing modes available on the 6502:

ADDRESSING MODE	EXAMPLE	HEX	BYTES
Immediate	LDA #5A0	A9	A0
Absolute	LDA \$7FA	AD	FA 07
Zero Page	LDA \$80	A4	80
Implicit/Implied	TAY	A8	
Relative	BCC \$3360	90	0F
Indexed	LDA \$200,X	BD	00 02
Indirect Indexed	LDA (\$80),Y	B1	80
Indexed Indirect	LDA (\$80,X)	A1	80

In looking at the examples, you should find all but the last three very familiar. We have used each of them in previous programs.

The *immediate mode* was used to load a register with a specific value. In most assemblers, this is indicated by the use of the number sign (#) preceding the value to be loaded. This contrasts with the *absolute mode* in which the value is retrieved from a given memory location. In this mode, the exact address you're interested in is given. *Zero page* is just a variation on the absolute mode. The main difference is the number of bytes used in the coding. It takes three in the general case; in zero page, only two are required.

Implicit, or *implied*, is certainly the most compact instruction in that only one byte is used. The TAY command, Transfer Accumulator to the Y-Register, needs no additional address bytes because the source and destination of the data are implied by the very instruction itself.

Relative addressing is done in relation to where the first byte of the instruction itself is found. Although the example interprets it as a branch to a specific address, you'll notice that the actual hex code is merely a plus or minus displacement from the branch point. This too was covered previously.

With these addressing modes, we can create quite a variety of programs. The problem with these modes is that the programs are rather inflexible to data from the outside world, such as those in input routines, and in doing things like accessing tables and large blocks of data.

Indexed Addressing

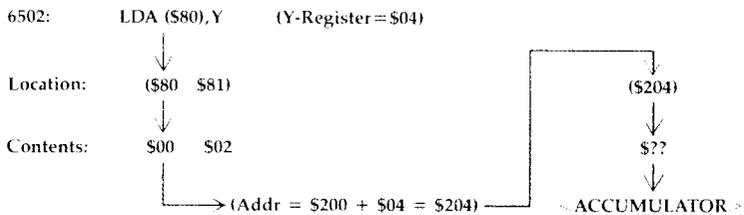
To access such data, we introduce the new idea of *indexed addressing*. In the pure form, the contents of the X-Register or Y-Register are added to the address given in the instruction to determine the final address. In the example given, if the X-Register holds a 0, the Accumulator will be loaded with the contents of location \$200. If, instead, the X-Register holds a 04, then location \$204 will be accessed. The usefulness in accessing tables and the like should be obvious.

The problem that arises here occurs when you want to access a table that grows or shrinks dynamically as the data within it changes. Another problem occurs when the table grows larger than 256 bytes. Because the maximum offset possible using the

X-Register or Y-Register is 255, we would normally be out of luck.

The solution to the byte limit is to use the *indirect indexed* mode. Indirect index is really an elegant method. First, the 6502 goes to the given zero page location (the base address *must* be on zero page). In the example, it would go to \$80 and \$81 to get the low-order and high-order bytes of the address stored there. Then it adds the value of the Y-Register to that address.

INDIRECT INDEXED ADDRESSING



Oftimes, these two-byte zero-page address pairs are called *pointers*, and you will hear them referred to in dealing with various programs on the Apple. In fact, by looking at pages 140 to 141 of the *Applesoft II Basic Reference Manual*, you will observe quite a number of these byte pairs used by Applesoft to keep track of all sorts of continually changing things, like where the program is, the locations of strings and other variables, and many nifty items.

If we wanted to simulate the LDA \$200, X command with the indirect mode, we would first store a #\$00 in \$80 and a #\$02 in \$81—00 and 02 being the low-order and high-order bytes of the address \$200. Then we'd use the command LDA (\$80), Y.

A much better (but unfortunately rarely used) term is *post-indexing*, referring to the fact that the index is added *after* the base address is determined.

Sometimes X and Y Aren't Interchangeable

You may have noticed that I used the X-Register in one case and the Y-Register in the other. It turns out that the X-Register and the Y Register cannot always be used interchangeably. The difference shows up depending on which addressing mode and what actual command you are using (LDA, STX, or others). As it

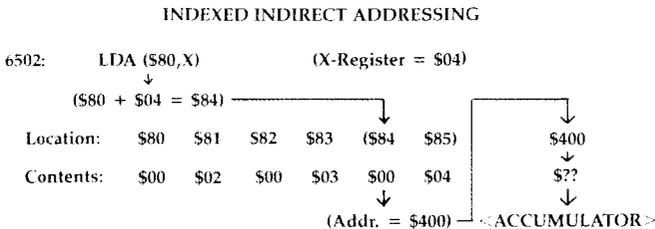
happens, indirect indexed addressing can only be done using the Y-Register.

To know which addressing modes can be used with a given command, you can refer to either of two appendices provided at the back of this book. Appendix B is rather like a dictionary of all the possible 6502 commands and devotes at least one page to each command. Appendix C on the other hand is a more condensed form of the first appendix, and may make it easier to compare available modes between a variety of commands.

I highly recommend making frequent use of Appendix B while you are learning machine language programming. It is essentially your toolbox of available commands for solving a particular programming problem. Whenever you're trying to write a particular routine and aren't sure just how to approach it, skim through this section of all possible commands and see if any particular command inspires you as to a possible approach. Granted, this is likely to happen more when you're working on rather simple goals such as moving a byte from here to there, but even the largest programs are made up of just such simple steps as that.

The last addressing mode, *indexed indirect*, is probably the most unusual. In this case, the contents of the X-Register (the Y-Register cannot be used for this mode) are added to the base address before going to get the contents. In a case similar to the other one, if the X-Register held 0, an LDA (\$80,X) would go to \$80 and \$81 for the two-byte address and then load the Accumulator with the contents of the indicated location. If, instead, the X-Register held a 04, the memory address would be determined by the contents of \$84 and \$85!

Usually, then, the X-Register is loaded with multiples of two to access a series of continuous pointers in zero page. This is also called *pre-indexing* since the index is added to the zero page location before determining the base address.



Storing Pure Data

Before we can put all this new information to work, we now need to answer one more question. How do you store just pure data within a program? All the commands we've covered so far are actual commands for the 6502. There is no data command as such. What are available, though, are the pseudo-ops, or assembler commands, of your particular assembler. These will vary from one assembler to another, so you'll have to consult your own manual to see how your assembler operates.

In general, the theory is to define a block of one or more bytes of data and then to skip over that block with a branch or jump instruction when executing your program. Usually, data can be entered either as hex bytes or as the ASCII characters you wish to use. In the second case, the assembler will automatically translate the ASCII characters into the proper hex numbers.

Most assemblers have a HEX command for directly entering the hex bytes of data table. The *DOS Tool Kit* assembler is one exception; using it, line 20 of the following listing should read, 20 DATA ASC DAPPLE. A sample program using the indexed address mode is given here:

```
1 *****
2 *SAMPLE DATA PROGRAM *
3 *****
4 *
5   ORG $300
6   OBJ $300
7 *
8   COUT EQU $FDED
9 *
10  START LDX #$00
11  LOOP LDA DATA,X
12    JSR COUT
13    INX
14    CPX #$05
15    BCC LOOP
16    LDA #$8D
17    JSR COUT
18  EXIT RTS
19 *
20  DATA HEX C1D0D0CC5
21 *
22 * DATA = 'APPLE'
```

When looked at in memory, it should appear like this:

*300L

```
0300-   A2 00       LDX   #$00
0302-   BD 0E 03   LDA   $030E,X
0305-   20 ED FD   JSR   $FDED
0308-   E8         INX
0309-   E0 05     CPX   #$05
030B-   90 F5     BCC   $0302
030D-   60       RTS
030E-   C1 D0     CMP   ($D0,X)
0310-   D0 CC     BNE   $02DE
0312-   C5 00     CMP   $00
```

This program is an improved version of the one we did earlier to print the word APPLE on the screen. It uses the indexed address mode to scan through the data table to print the word APPLE. Notice that data tables may be wildly interpreted to the screen when disassembling. This is because the Apple has no way of knowing what part of the listing is data and tries to list data as a usual machine language program.

Basically, the idea of the program is to loop through, getting successive items from the data table using the offset of the X-Register. When the X-Register reaches 05 (the number of items in the table), we are finished printing. After printing, we terminate with a carriage return. Remember that in machine language we must usually do everything ourselves. This means we cannot assume an automatic carriage return at the end of a printed string.

Note that the hex values in the data table are the ASCII values for each letter plus \$80. This sets the high bit of each number, which is what the Apple expects in order to have the letter printed out properly when using COUT.

The indirect addressing modes are used when you want to access in a very compact and efficient way. Let's consider the problem of clearing the screen, for instance. We want to put a space character in every memory location in the screen block (\$400-\$7FF). Here is one way of doing this:

```
1 *****
2 * SCREEN CLEAR PROG. 1A *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PTR EQU $06
9 *
```

```

10 ENTRY LDA #$04
11 STA PTR +1
12 LDY #$00
13 STY PTR
14 * SETS PTR (6,7) TO $400
15 START LDA #$A0
16 LOOP STA (PTR),Y
17 INY
18 BNE LOOP
19 NXT INC PTR +1
20 LDA PTR +1
21 CMP #$08
22 BCC START
23 EXIT RTS

```

Listed from the Monitor, it should appear like this:

```

*300L

0300-  A9 04      LDA  #$04
0302-  85 07      STA  $07
0304-  A0 00      LDY  #$00
0306-  84 06      STY  $06
0308-  A9 A0      LDA  #$A0
030A-  91 06      STA  ($06),Y
030C-  C8         INY
030D-  D0 FB      BNE  $030A
030F-  E6 07      INC  $07
0311-  A5 07      LDA  $07
0313-  C9 08      CMP  #$08
0315-  90 F1      BCC  $0308
0317-  60         RTS

```

We start off by initializing locations \$06 and \$07 to hold the base address of \$400, the first byte of the screen memory area. Then we enter a loop that runs the Y-Register from \$00 to \$FF. Since this is added to the base address in \$06,07, we then store a \$A0 (a space) in every location from \$400 to \$4FF. When Y is incremented from \$FF, it goes back to \$00, and this is detected by the BNE on line 18. At zero, it falls through and location \$07 is incremented from \$04 to \$05, giving a new base address of \$500. This whole process is repeated until location \$07 reaches a value of \$08 (corresponding to a base address of \$800), at which point we return from the routine.

By changing the value of the #\$A0 to some other character, we can clear the screen to any character we wish. In fact, you can get the value from the keyboard as we've done in earlier programs.

Here is a revised version:

```
1 *****
2 * SCREEN CLEAR PROG. 1B *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PTR EQU $06
9   CHAR EQU $08
10  KYBD EQU $C000
11  STROBE EQU $C010
12 *
13  ENTRY LDA #$04
14    STA PTR +1
15    LDY #$00
16    STY PTR
17 * SETS PTR (6,7) TO $400
18  READ LDA KYBD
19    CMP #$80; KEYPRESS?
20    BCC READ; NO, THEN TRY AGAIN.
21    STA STROBE; CLEAR KYBD STROBE.
22    STA CHAR
23  CLEAR LDY #$00
24    LDA CHAR
25  LOOP STA (PTR),Y
26    INY
27    BNE LOOP
28  NXT INC PTR +1
29    LDA PTR +1
30    CMP #$08
31    BCC CLEAR
32  AGAIN JMP ENTRY
```

It should appear like this in listed form:

*300L

```
0300-   A9 04       LDA   #$04
0302-   85 07       STA   $07
0304-   A9 00       LDY   #$00
0306-   84 06       STY   $06
0308-   AD 00 C0    LDA   $C000
030B-   C9 80       CMP   #$80
030D-   90 F9       BCC   $0308
030F-   8D 10 C0    STA   $C010
0312-   85 08       STA   $08
0314-   A0 00       LDY   #$00
0316-   A5 08       LDA   $08
0318-   91 06       STA   ($06),Y
```

```

031A-  C8          INY
031B-  D0 FB      BNE  $0318
031D-  E6 07      INC  $07
031F-  A5 07      LDA  $07
0321-  C9 08      CMP  #$08
0323-  90 EF      BCC  $0314
0325-  4C 00 03   JMP  $0300

```

Enter this program and run from Basic with a CALL 768. Each press will clear the screen to a different character. The screen should clear to the same character as the key you press, including space bar and special characters. In this program especially, you can see how fast machine language is. To clear the screen requires loading more than one thousand different locations with the given value. In Applesoft, this process would be quite slow by comparison. In assembly language, you'll find that the screen will clear to different characters just as fast as you can type them.

An interesting variation on this is to enter the graphics mode by typing in GR before calling the routine. Then the screen will clear to various colors and different line patterns.

In this variation on program 1A we've used the principles from chapter five where we read the keyboard until we got a value greater than \$80, meaning a key has been pressed. This value is held temporarily in the variable CHAR so that it can be retrieved each time after incrementing the PTR in the NXT section.

See what variations you can make on this, or try the hi-res screen (\$2000 through \$3FFF).

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25



CHAPTER 7

Sound Generation Routines

Sound generation in machine language is such a large topic in itself, that an entire book could be done on that subject alone. However, simple routines are so easy that they're worth at least a brief examination here. These routines will not only allow you to put the commands you've learned to further use, but are also just plain fun.

The first element of a sound generating routine is the speaker itself. Recall that the speaker is part of the memory range, from \$C000 to \$C0FF, which is devoted entirely to hardware items of the Apple II. In earlier programs, we looked at the keyboard by examining memory location \$C000. The speaker can be similarly accessed by looking at location \$C030. The exception here is that the value at \$C000 (the keyboard) varied according to what key was pressed, whereas with \$C030 (the speaker) there is no logical value returned.

Every time location \$C030 is accessed, the speaker will click once. This is easy to demonstrate. Simply enter the Monitor with a `CALL - 151`. Enter `C030` and press `RETURN`. You'll have to listen carefully, and you may have to try it several times. Each time, the speaker will click once. You can imagine that, if we could repeatedly access the speaker at a fast enough rate, the series of clicks would become a steady tone. In Basic, this can be done, although poorly, by a simple loop such as this:

10 X = PEEK (-16336): GOTO 10

The pitch of the tone generated depends on the rate at which the speaker is accessed. Because Integer Basic is faster in its execution than Applesoft, the tone generated will be noticeably higher in pitch in the Integer version.

In machine language, the program would look like this:

```
0300- AD 30 C0 LDA $C030
0303- 4C 00 03 JMP $0300
```

In this case, I'm showing it as the Apple would directly disassemble it as opposed to the usual assembly language source listing. The program is so short that the easiest way to enter it is by typing in the hex code directly. To do this, enter the Monitor (CALL -151) and type:

```
300: AD 30 C0 4C 00 03
```

Then run the program by typing 300G.

Disappointed? The program is working. The problem is that the routine is actually too fast for the speaker to respond. What's lacking here is some way of controlling the rate of execution of the loop. This is usually accomplished by putting a delay of some kind in the loop. We should also be able to specify the length of the delay, either before the program is run or, even better, during the execution of the program.

Delays

We can do this any of three ways: 1) physically alter the length of the program to increase the execution time of each pass through the loop; 2) store a value somewhere in memory before running the program and then use that value in a delay loop; or 3) get the delay value on a continual basis from the outside world, such as from the keyboard or paddles.

For the first method, you can use a new and admittedly complex command. The mnemonic for this instruction is NOP and stands for No Operation. Whenever the 6502 microprocessor encounters this, it just continues to the next instruction without doing anything. This code is used for just what we need here—a time delay.

It is more often used, though, as either a temporary filler when assembling a block of code (such as for later data tables) or to cancel out existing operations in a previously written section of code. Quite often, this code (\$EA, or 234 in decimal) is seen being used in this manner to patch parts of the Apple DOS to cancel out various features that you no longer want to have active (such as the NOT DIRECT command error that prevents you from doing a GOTO directly to a line that has a DOS command on it).

In our sound routine, an NOP will take a certain amount of time even to pass over and will thus reduce the number of cycles per second of the tone frequency. The main problem in writing the new version will be the number of NOPs that will have to be inserted.

The easiest way to get a large block of memory cleared to a specific value is to use the move routine already present in the Monitor. To clear the block, load the first memory location in the range to be cleared with the desired value. Then type in the move command, moving everything from the beginning of the range to the end up one byte. For instance, to clear the range from \$300 to \$3A0 and fill it with \$EAs, you would, from the Monitor, of course, type in:

```
300: EA
301<300.3A0M
```

Note that we are clearing everything from \$300 to \$3A0 to contain the value \$EA.

Now type in:

```
300: AD 30 C0
3A0: 4C 00 03
```

Then type in 300L, followed with L for each additional list section, to view your new program.

```
*300L
```

```
0300- AD 30 C0 LDA $C030
0303- EA NOP
0304- EA NOP
0305- EA NOP
0306- EA NOP
0307- EA NOP
```

```

0308- EA      NOP
0309- EA      NOP
*      *      *
*      *      *
*      *      *
0395- EA      NOP
0396- EA      NOP
0397- EA      NOP
0398- EA      NOP
0399- EA      NOP
039A- EA      NOP
039B- EA      NOP
039C- EA      NOP
039D- EA      NOP
039E- EA      NOP
039F- EA      NOP
03A0- 4C 00 03  JMP  $0300

```

Now run this with the usual 300G.

The tone produced should be a very nice, pure tone. The pitch of the tone can be controlled by moving the JMP \$300 to points of varying distance from the LDA \$C030. Granted, this is a very clumsy way of controlling the pitch and is rather permanent once created, but it does illustrate the basic principle.

For a different tone, hit RESET to stop the program, then type in: 350:4C 00 03. When this is run (300G), the tone will be noticeably higher. The delay time is about half of what it was, and thus the frequency is twice the original value. Try typing in the three bytes in separate runs at \$320 and \$310. At \$310, you may not be able to hear the tone, because the pitch is now essentially in the ultrasonic range.

I think you'll also notice that all these tones are of a very pure nature and, in general, much nicer than those generated by a Basic program.

Delay Value In Memory

Usually the way tone programs work is to pass a pitch value from Basic by putting the value in a memory location. This program is an example of that technique.

```

1 *****
2 * SOUND ROUTINE 2 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8 *
9   PITCH EQU $06
10  SPKR EQU $C030
11 *
12  ENTRY LDY PITCH
13   LDA SPKR
14  LOOP DEY
15   BNE LOOP
16   JMP ENTRY
17 * INF. LOOP

```

When assembled, it should look like this:

***300L**

```

0300-   A4 06           LDY   $06
0302-   AD 30 C0       LDA   $C030
0305-   88             DEY
0306-   D0 FD           BNE   $0305
0308-   4C 00 03       JMP   $0300

```

In this program, we get a value of \$00 to \$FF from location \$06 (labeled *pitch*) and put it in the Y-Register. The speaker is then clicked. At that point, we enter a delay loop that cycles n times where n is the number value for pitch held in location \$06.

To run this program, first load location \$06 with values of your choice (0 to 255 decimal, \$00 to \$FF hex) and then run the 300G. This is more compact and controllable than the first example, but it still suffers from being an infinite loop. What we need to do is specify a duration for the tone as well. Again you have the option of either making the value part of the program or passing it in the same way as we're currently doing the value for pitch. Here's a listing for the new program:

```

1 *****
2 * SOUND ROUTINE 3 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8 *
9   PITCH EQU $06
10  DURATION EQU $07
11  SPKR EQU $C030
12 *
13  ENTRY LDX DURATION
14  LOOP LDY PITCH
15    LDA SPKR
16    DELAY DEY
17    BNE DELAY
18  DRTN DEX
19    BNE LOOP
20  EXIT RTS

```

Disassembled, it will appear like this:

*300L

```

0300-   A6 07       LDX   $07
0302-   A4 06       LDY   $06
0304-   AD 30 C0    LDA   $C030
0307-   88         DEY
0308-   D0 FD       BNE   $0307
030A-   CA         DEX
030B-   D0 F5       BNE   $0302
030D-   60         RTS

```

This routine is used by loading \$06 with a value for the pitch you desire, \$07 with a value for how long you want the tone to last, and then running the routine with the 300G.

Examining this listing, you'll see that it is basically an extension of routine 2. In this revised version, instead of always jumping back to the LDY of the play cycle, we decrement a counter (the X-Register). This counts the number of times we're allowed to pass through the loop, and thus the final length of the play.

This can be used by Basic programs, as illustrated by this sample Applesoft listing:

```

10 PRINT CHR$(4);"BLOOD TONE ROUTINE,A$300"
20 INPUT"PITCH, DURATION?";P,D
30 POKE6,P:POKE7,D
40 CALL 768
50 PRINT
60 GOTO 20

```

This makes it very easy to experiment with different values for both pitch and duration. The main bug in this routine is that even for a fixed value for duration, the length of the note will vary depending on the pitch specified. This is because less time spent in the delay loop means less overall execution time for the routine as a whole.

Delay from Keyboard or Paddles

The next variation is to get the pitch on a continual basis from an outside source. In this case, the source will be the keyboard. Type in and assemble this source listing:

```

1 *****
2 * SOUND ROUTINE 4 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8 *
9   KYBD EQU $C000
10  STROBE EQU $C010
11  SPKR EQU $C030
12 *
13  ENTRY LDA KYBD
14   STA STROBE
15   CMP #$80
16   BEQ EXIT
17   TAY
18  LOOP LDA SPKR
19  DELAY DEY
20   BNE DELAY
21   JMP ENTRY
22  EXIT RTS

```

In memory, it should look like this:

***300L**

```
0300-   AD 00 C0   LDA   $C000
0303-   8D 10 C0   STA   $C010
0306-   C9 80     CMP   #$80
0308-   F0 0A     BEQ   $0314
030A-   A8       TAY
030B-   AD 30 C0   LDA   $C030
030E-   88       DEY
030F-   D0 FD     BNE   $030E
0311-   4C 00 03   JMP   $0300
0314-   60       RTS
```

Running this will give you a really easy way of passing tone values to the routine. Characters with low ASCII values will produce higher tones than ones with higher values. This means that the control characters will produce unusually high tones. To exit press control-shift-P (control-@).

Let's review how the routine functions.

At the entry point (\$300), the very first thing done is to get a value from the keyboard. The strobe is then cleared, and an immediate check done for #\$80. Remember that \$80 is added to the ASCII value for each key pressed when read at \$C000. A value less than \$80 means no key has been pressed. Checking specifically for \$80, the computer looks to see if a control-@ has been pressed. This is just a nice touch to give us a way of exiting the program. After the check, we transfer the Accumulator value (equivalent to pitch in the earlier programs) to the Y-Register and finish with the same routine used in sound routine 2.

Of course, I have to give you at least one program using the paddles. This one gives us an opportunity to use the external routines in the Monitor, too. Don't forget that using the routines already present in the Monitor is the real secret to easy machine language programming. It saves you the trouble of having to write your own I/O and other sophisticated routines and lets you concentrate on those aspects unique to your program.

Now for the program:

```
1 *****
2 * SOUND ROUTINE 5 *
3 *****
4 *
5 *
6   OBJ $300
```

```

7   ORG $300
8   *
9   PDL EQU $FB1E
10  SPKR EQU $C030
11  *
12  ENTRY LDX #$00
13   JSR PDL
14   LDA SPKR
15   LDX #$01
16   JSR PDL
17   LDA SPKR
18   JMP ENTRY
19  * INF. LOOP

```

The Monitor will list this as:

```

*300L

0300-   A2 00           LDX   #$00
0302-   20 1E FB       JSR   $FB1E
0305-   AD 30 C0       LDA   $C030
0308-   A2 01           LDX   #$01
030A-   20 1E FB       JSR   $FB1E
030D-   4C 00 03       JMP   $0300

```

Running this should produce some really interesting results. The theory of this routine is elegantly simple. It turns out that just reading a paddle takes a certain amount of time, sufficient to create our needed delay. The greater the paddle reading, the longer the delay to read it.

What happens in this routine is that we actually have two distinct delays created, one by each paddle. Remember that, to read a paddle, you first have to load the X-Register with the number of the paddle you wish to read and then do the JSR to the paddle read routine. The value is returned in the Y-Register, but in this case we don't need to know what the value was.

The combination of the two different periods of delay creates the effect of two tones at once and a number of other very unique sounds.

This has been only the most basic discussion of sound generating in machine language, but I think you'll find that it illustrates what can be done with only a few commands, and that machine language offers many advantages in terms of memory use and execution speed.

111

112

113

114

115

116

117

118

119

120

121

122

123

124

125



CHAPTER 8

The Stack

One of the more obscure parts of the operation of the Apple is related to something called the *stack*. This is a part of memory reserved for holding return addresses for GOSUBS and FOR-NEXT loops, and a few other operations in direct machine code.

If you want to impress your friends with your knowledge of machine language, just throw this term around in a confident manner and they'll figure you must be an expert!

The stack can be thought of like those spring-loaded plate holders they have in restaurants. Plates are loaded onto the top of a cylinder with a spring-loaded platform in it. As more plates are added, the bottom one gets pushed down. The plates must always be removed in the opposite order from that in which they are put in. The catch phrase for this is LIFO, for Last-In, First-Out. The first location loaded in the 6502 stack is \$1FF. Rather than pushing everything down toward \$100 each time a new value is put on the stack, the 6502 has a stack pointer that is adjusted as new data is added. Successive values are added in descending order, with the stack pointer being reset each time to indicate the position of the last value put in. Thus the table is created in reverse order, building downward.

The technical details of its operation are not required to make good use of the stack. One of the most convenient things the stack can be used for is to hold values temporarily while you're doing something else. Normally, in a program, we'd have to

assign a zero page location to hold a value. For instance, consider this program:

```
1 *****
2 * BYTE DISPLAY PROG. 1 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   CHR EQU $06
9   PRBYTE EQU $FDDA
10  COUT EQU $FDED
11  PREAD EQU $FB1E
12  HOME EQU $FC58
13 *
14  START JSR HOME
15  GETCHR LDX #$00
16   JSR PREAD
17   STY CHR
18   TYA
19   JSR PRBYTE
20   LDA #$A0 ; SPACE
21   JSR COUT
22   LDA CHR
23   JSR COUT
24   LDA #$8D ; RETURN
25   JSR COUT
26   JMP GETCHR
```

This will be listed by the Monitor as:

```
*300L

0300- 20 58 FC   JSR   $FC58
0303- A2 00     LDX   #$00
0305- 20 1E FB   JSR   $FB1E
0308- 84 06     STY   $06
030A- 98       TYA
030B- 20 DA FD   JSR   $FDDA
030E- A9 A0     LDA   #$A0
0310- 20 ED FD   JSR   $FDED
0313- A5 06     LDA   $06
0315- 20 ED FD   JSR   FDED
0318- A9 8D     LDA   #$8D
031A- 20 ED FD   JSR   $FDED
031D- 4C 03 03   JMP   $0303
```

This program gets a value from \$00 to \$FF from paddle zero, and stores it in location \$06. This is needed because the JSR to \$FDDA (a handy routine that prints the hex number in the Accumulator) scrambles the Accumulator and Y-Register. We want to keep the value at hand because the ASCII character corresponding to it is then printed out right after the number using COUT. The cycle then repeats until you press RESET

Location \$06 is used for only a moment each pass to store the value temporarily. In addition, it commits that zero page location to use and thus limits our choices when we need other places to store something. A better system is to make use of the stack. The commands to do this are PHA and PLA. PHA stands for PusH Accumulator onto stack. When this is used in line 17 below, the value currently in the Accumulator is put onto the stack. The Accumulator itself goes unaltered, and none of the status flags, such as the carry or zero flags, are conditioned. The value is simply copied and stored for us.

Later on, when we want to retrieve the value, the PLA on line 21 (for PulL Accumulator from stack) pulls the value back off the stack into the Accumulator. A PLA command does condition the zero flag, and also the sign bit, which has not been covered yet.

Important: For each PHA there must be a PLA executed before encountering the next RTS in a program.

Here's the revised program:

```
1 *****
2 * BYTE DISPLAY PROG. 2 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8 PRBYTE EQU $FDDA
9 COUT EQU $FDED
10 PREAD EQU $FB1E
11 HOME EQU $FC58
12 *
13 START JSR HOME
14 GETCHR LDX #$00
15   JSR PREAD
16   TYA
17   PHA
18   JSR PRBYTE
19   LDA #$A0 ; SPACE
20   JSR COUT
```

```

21  PLA
22  JSR COUT
23  LDA #$8D ; RETURN
24  JSR COUT
25  JMP GETCHR

```

This will list like so:

*300L

```

0300- 20 58 FC   JSR   $FC58
0303-  A2 00     LDX   #$00
0305- 20 1E FB   JSR   $FB1E
0308-  98       TYA
0309-  48       PHA
030A- 20 DA FD   JSR   $FDDA
030D-  A9 A0     LDA   #$A0
030F-  68       PLA
0310- 20 ED FD   JSR   $FDED
0313-  A9 8D     LDA   #$8D
0315- 20 ED FD   JSR   $FDED
0318-  4C 03 03  JMP   $0303

```

The stack is also used automatically by the 6502 for storing the return address for each JSR as it's encountered. Each time you do a PHA, this address is buried one level deeper. You must have done an equivalent number of PLAs at some point in the routine before reaching the next RTS to have things work properly.

Also remember, if you want to store more than one value, you must retrieve the values in the opposite order in which they were stored. Once a value is removed from the stack with the PLA, it is essentially gone forever from the stack unless you put it back directly.

There is a limit to how much you can put in the stack. The limit of sixteen GOSUBS and FOR-NEXT loops in Basic is related to the use of this. Technically you can put 256 one-byte values, or 128 RTS addresses on the stack, but the Apple also uses it for its own operations, and, many times, you have Basic going, too.

In general, though, it rarely fills up unless you're getting extreme in its use, and at that point the code probably will be so tangled in nested subroutines that you may want to consider a rewrite anyway!

Try using the stack in some of your own programs; I think you'll find it quite useful.



CHAPTER 9

Math Operations

Now let's look at the simple math operations of addition and subtraction in machine language. To an extent, we've already done some of this. The increment and decrement commands (INC/DEC, and so on) add and subtract for us. Unfortunately, they only do so by one each time ($\text{VALUE} + 1$ or $\text{VALUE} - 1$).

If you're really ambitious, you could, with the commands you have already, add or subtract any number by using a loop of repetitive operations, but this would be a bit tedious, not to mention slow. Fortunately, a better method exists. But, first, a quick review of some binary math facts.

In chapter three, we discussed the idea behind binary numbers and why they're so important in computers. I would like to further elaborate on the topic now and show how the idea of binary numbers applies to basic arithmetic operations in machine language programming.

Binary Numbers

By now you're certainly comfortable with the idea of a byte being an individual memory location which can hold a value from \$00 to \$FF (0 to 255). This number comes about as a direct result of the way the computer is constructed and the way in which you count in base two.

Each byte can be thought of as being physically made up of

eight individual switches that can be in either an *on* or *off* position. We can count by assigning each possible combination of ons and offs a unique number value.

In the following diagrams, if a switch is off, it will be represented by a 0 in its particular position. If it's on, a 1 will be shown. When all the switches are off, we'll call that zero.

In base two, each of the eight positions in the byte is called a bit, and the positions are numbered from right to left, from 0 to 7.

The pattern for counting is similar to normal decimal or hex notation. The value is increased by adding one each time to the digit on the far right, *carrying* as necessary. In base ten, you'd have to carry every tenth count, in hex, every sixteenth. In base two, the carry will be done *every other time!*

So . . . the first few numbers look like this:

Hex	Decimal	Binary
\$00	0	0 0 0 0 0 0 0 0
\$01	1	0 0 0 0 0 0 0 1
\$02	2	0 0 0 0 0 0 1 0
\$03	3	0 0 0 0 0 0 1 1
\$04	4	0 0 0 0 0 1 0 0

Notice that in going from the value one to the value two, we would add a 1 to the 1 already at the first position (bit 0). This generates the carry to increment the second position (bit 1). Here is the end of the series:

\$FD	253	1 1 1 1 1 1 0 1
\$FE	254	1 1 1 1 1 1 1 0
\$FF	255	1 1 1 1 1 1 1 1

Now the most important part. Observe what happens when the upper limit of the counter is finally reached. At \$FF (255), all positions are *full*. When the next increment is done, we should carry a one to the next position to the left; unfortunately, that next position doesn't exist!

Addition with ADC

This is where the carry bit of the Status Register is used again. Before, it was used in the compare operations (CMP, for instance), but, as it happens, it is also conditioned by the next

command, ADC. This stands for ADd with Carry. When the next step is done using an ADC command, things will look like this:

										Carry
\$100	256	0	0	0	0	0	0	0	0	1

The byte has returned to a value of zero and the carry bit is set to a one.

We discussed the wrap-around to zero earlier, with the increment/decrement commands, but we didn't mention the carry. That's because the INC/DEC commands don't affect the carry flag.

However, the ADC command *does* condition the carry flag. The carry will be set whenever the result of the addition is greater than \$FE. With ADC, you can make your counters increment by values other than one—rather like the FOR I = 1 TO 10 STEP 5 statement in Basic. But ADC is used more often for general math operations, such as calculating new addresses or screen positions, among a wide variety of other applications.

Whenever ADC is used, the value indicated is added to the contents of the Accumulator. The value can be stated either directly by use of an immediate value or with an indirect value.

Important Note: Although the ADC conditions the carry after it is executed, it cannot be assumed that the carry is conveniently standing in a *clear* condition when the addition routine is begun.

For example, consider this simple program:

```
LDA #$05
ADC #$00
STA RESULT
```

As it stands, there are two possible results. If the carry happened to be clear when this was executed, the value in RESULT would be \$05. If, however, the carry had been set (perhaps as the result of some other operation), then RESULT would have been \$06.

The point of all this is that the carry flag must be cleared before the first ADC operation. The example above should have been written as:

```
CLC (Clear Carry)
LDA #$05
```

ADC #\$00
STA RESULT

In this case, RESULT will always end up holding the value \$05.

Here are some sample programs for using the ADC. Note the use of the CLC before each ADC.

```
1 *****
2 * SAMPLE PROGRAM 1 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   N1 EQU $06
9   N2 EQU $08
10  RSLT EQU $0A
11 *
12  START LDA N1
13    CLC
14    ADC N2
15    STA RSLT
16  END RTS
```

```
1 *****
2 * SAMPLE PROGRAM 2 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   N1 EQU $06
9   RSLT EQU $0A
10 *
11 *
12  START LDA N1
13    CLC
14    ADC #$80
15    STA RSLT
16  END RTS
```

```
1 *****
2 * SAMPLE PROGRAM 3 *
3 *****
4 *
5   OBJ $300
6   ORG $300
```

```

7 *
8 N1 EQU $06
9 INDX EQU $08
10 RSLT EQU $0A
11 TBL EQU $300
12 *
13 START LDA N1
14   LDX INDX
15   CLC
16   ADC TBL,X
17   STA RSLT
18 END RTS

1 *****
2 * SAMPLE PROGRAM 4 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8 N1 EQU $06
9 INDX EQU $08
10 RSLT EQU $0A
11 PTR EQU $1E
12 *
13 START LDA #$00
14   STA PTR
15   LDA #$03
16   STA PTR + 1
17   LDA N1
18   LDY INDX
19   CLC
20   ADC (PTR), Y
21   STA RSLT
22 END RTS

```

In the first program, the value in N1 is added to the contents of N2. In the second, N1 is added to the immediate value \$80. Note the CLC before the ADC to ensure an accurate result. This result is then returned in location \$0A. This routine could be used as a subroutine for another machine language program, or it could be called from Basic after passing the values to locations \$06 and \$08.

The latter two programs are more elaborate examples where the indirect modes are used to find the value from a table starting at \$300. In program 3, an index value is passed to location \$08. That is used as an offset via the X-Register. In program 4, the low-order and high-order bytes for the address \$300 are first put

in a pair of pointer bytes (\$1E,\$1F) and the offset is put in the Y-Register.

In all the programs, however, we are limited to adding two single-byte values, and further restricted to a one-byte result. Not very practical in the real world.

The solution is to use the carry flag to create a two-byte addition routine. Here's an example:

```
1 *****
2 * SAMPLE PROGRAM 5A *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   N1 EQU $06
9   N2 EQU $08
10  RSLT EQU $0A
11 *
12  START CLC
13   LDA N1
14   ADC N2
15   STA RSLT
16   LDA N1+1
17   ADC N2+1
18   STA RSLT+1
19  END RTS
```

*300L

```
0300- 18    CLC
0301-  A5 06 LDA    $06
0303-  65 08 ADL    $08
0305-  85 0A STA    $0A
0307-  A5 07 LDA    $07
0309-  65 09 ADC    $09
030B-  85 0B STA    $0B
030D-  60    RTS
```

Notice that N1, N2, and RSLT are all two-byte numbers, with the second byte of each pair being used for the high-order byte. (If you're unsure of the idea of low and high-order bytes, refer to chapter one (footnote two). This allows us to use values and results from \$00 to \$FFFF (0 to 65535). This is sufficient for any address in the Apple II, although, by using three or more bytes, we could accommodate numbers much larger than \$FFFF.

A few words of explanation about this program. First, the

CLC has been moved to the beginning of the routine. Although it need only precede the ADC command, it has no effect on the LDA, so it is put at the beginning of the routine for aesthetic purposes. Once the two low-order bytes of N1 and N2 are added and the partial result stored, the high-order bytes are added. If the result of this first addition is greater than 255, the carry will be set and an extra unit added in the second addition. Note that the carry remains unaffected during the LDA N1 + 1 operation.

The Monitor listing is given for this one so that you can enter it and then call it from the Basic program:

List

```

0 REM MACHINE ADDITION ROUTINE
10 HOME
20 INPUT "N1,N2?";N1,N2
30 N1 = ABS (N1): N2 = ABS (N2)
40 POKE 6, N1 - INT (N1/256) * 256: POKE 7, INT (N1/256) *
   256:POKE 9, INT (N2/256)
50 POKE 8, N2 - INT (N2/256) * 256: POKE 9, INT (N2/256)
60 CALL 768
70 PRINT: PRINT "RESULT IS"; PEEK (10) + 256 * PEEK (11)
80 PRINT:GOTO 20

```

The ABS() statements on line 30 eliminate values less than zero. Although there are conventions for handling negative numbers, this routine is not that sophisticated.

Many times, the number being added to a base address is known always to be \$FF or less, so only one byte for N2 is needed. A two/one addition routine looks like this:

```

1 *****
2 * SAMPLE PROGRAM 5B *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   N1 EQU $06
9   N2 EQU $08
10  RSLT EQU $0A
11 *
12  START CLC
13   LDA N1
14   ADC N2
15   STA RSLT

```

```

16  BCC END
17  LDA N1+1
18  ADC #$00
19  STA RSLT+1
20  END RTS

```

```

1  *****
2  * SAMPLE PROGRAM 5C *
3  *****
4  *
5  OBJ $300
6  ORG $300
7  *
8  N1 EQU $06
9  N2 EQU $08
10 RSLT EQU $0A
11 *
12 START CLC
13 LDA N1
14 ADC N2
15 STA RSLT
16 BCC END
17 LDA N+1
18 STA RSLT+1
19 INC RSLT+1
20 END RTS

```

For speed, if a carry isn't generated on line 14, the program skips directly to the end. If, however, the carry is set, the value in `N1+1` gets incremented by one, even though the `ADC` says an immediate `$00`. The `$00` acts as a dummy value to allow the carry to do its job. If speed is not a concern, the `BCC` can even be left out with no ill effect. Program 5C shows an alternate method using the `INC` command. In this case, the `BCC` is required for proper operation.

The reason for bringing up listing 5C is that the most common reason for adding one to a two-byte number is to increment an address pointer by one. In that case, the result is usually put right back in the original location, rather than in a separate `RESULT`. A routine for this is more compact and would look like this:

```

1  *****
2  * SAMPLE PROGRAM 5D *
3  *****
4  *

```

```

5   OBJ $300
6   ORG $300
7   *
8   N1 EQU $06
9   N2 EQU $08
10  RSLT EQU $0A
11  *
12  START CLC
13  INC N1
14  BCC END
15  INC N1+1
16  END RTS

```

Subtraction

Subtraction is done like addition except that a *borrow* is required. Rather than using a separate flag for this operation, the computer recognizes the carry as sort of a reverse borrow.

That is, a set carry flag will be treated by the subtract command as a *clear borrow* (no borrow taken); a clear carry as a *set borrow* (borrow unit taken).

The command for subtraction is *SBC*, for *SuBtract with Carry*. The borrow is cleared with the command *SEC*, for *SEt Carry*. (Remember, things are backward here). A subtraction equivalent of program 5A looks like this:

```

1  *****
2  * SAMPLE PROGRAM 6 *
3  *****
4  *
5  OBJ $300
6  ORG $300
7  *
8  N1 EQU $06
9  N2 EQU $08
10 RSLT EQU $0A
11 *
12 START SEC
13 LDA N1
14 SBC N2
15 STA RSLT
16 LDA N1+1
17 SBC N2+1
18 STA RSLT+1
19 END RTS

```

The program can be called with the same Basic program we used for the addition routine (program 5A).

Positive and Negative Numbers

So far, we have only discussed how to represent whole numbers greater than or equal to zero, using one or two bytes. A reasonable question then is: "How do we represent negative numbers?"

Negative numbers can be thought of as a way of handling certain common arithmetic possibilities, such as when subtracting a larger number from a smaller one, such as in $3 - 5 = -2$, and when adding a positive number to a negative number to obtain a given result, such as in $5 + -8 = -3$.

To be successful, then, what we must come up with is a system using the eight bits in each byte that will be consistent with signed arithmetic as we are currently familiar with.

The Sign Bit

The most immediate solution to the question of signed numbers is to use bit 7 to indicate whether a number is positive or negative. If the bit is clear, the number is positive. If the bit is set, the number will be regarded as negative.

Thus +5 would be represented:

00000101

While -5 would be shown as:

10000101

Note that by sacrificing bit 7 to show the sign, we're now limited to values from -127 to +127. When using two bytes to represent a number such as an address, this means that we'll be limited to the range of -32767 to +32767. Sound familiar? If you've had any experience with Integer Basic, then you'll recognize this as the maximum range of number values within that language.

Although this new scheme is very pleasing in terms of simplicity, it does have one minor drawback—it doesn't work. If we attempt to add a positive and negative number using this scheme we get disturbing results:

$$\begin{array}{r}
 5 \quad 00000101 \\
 + \quad -8 \quad 10001000 \\
 \hline
 -3 \quad 10001101 = -13!
 \end{array}$$

Although we should get -3 as the result, using our signed bit system we get -13 . Tsk, tsk. There must be a better way. Well, with the help of what is essentially a little numeric magic, we can get something which works, although some of the conceptual simplicity gets lost in the process.

What we'll invoke is the idea of number *complements*. The simplest complement is what is called a *ones' complement*. The ones' complement of a number is obtained by reversing each one and zero throughout the original binary number.

For example, the ones' complement to 5 would be:

$$\begin{array}{l}
 00000101 = 5 \\
 11111010 = -5
 \end{array}$$

For 8, it would be:

$$\begin{array}{l}
 00001000 = 8 \\
 11110111 = -8
 \end{array}$$

This process is essentially one of *definition*, that is to say that we declare to the world that 11110111 will now represent -8 without specifically trying to justify it. Undoubtedly there are lovely mathematical proofs of such things that present marvelous ways of spending an afternoon, but for our purpose, a general notion of what the terms mean will be sufficient. Fortunately computers are very good at following arbitrary numbering schemes without asking "but why is it that way?"

Now let's see if we're any closer to a working system:

$$\begin{array}{r}
 5 \quad 00000101 \\
 + \quad -8 \quad 11110111 \\
 \hline
 -3 \quad 11111100 = -3 \\
 (00000011 = +3)
 \end{array}$$

Hmmm . . . Seems to work pretty good. Let's try another:

$$\begin{array}{r}
 -5 \quad 11111010 \\
 + \quad 8 \quad 00001000 \\
 \hline
 3 \quad 00000010 = 2 \\
 \text{(Plus Carry)}
 \end{array}$$

Well our answers will be right half the time . . . It turns out there is a final solution, and that is to use what is called the *twos' complement* system.

The only difference between this and the ones' complement system is that after deriving the negative number by reversing each bit of its corresponding positive number, *we add one*.

Sound mysterious. Let's see how it looks:

<p>For -5: $5 = 0000101$ \downarrow 1111010 \downarrow $-5 = 1111011$</p>	<p>ones' complement . . .</p> <p>now add one . . .</p>	<p>For -8: $8 = 00001000$ \downarrow 11110111 \downarrow $-8 = 11111000$</p>
---	--	--

Now let's try the two earlier operations:

$$\begin{array}{r}
 5 \quad 0000101 \\
 + \quad -8 \quad 11111000 \\
 \hline
 -3 \quad 11111101 = -3
 \end{array}$$

Does 11111101
equal -3?

sample #: (00000011 = 3)

ones' comp: (11111100)
(+ 1)

twos' comp: (11111101 = -3) ✓

$$\begin{array}{r}
 -5 \quad 11111011 \\
 + \quad 8 \quad 00001000 \\
 \hline
 3 \quad 00000011 = 3
 \end{array}$$

(Plus Carry)

At last! It works in both cases. It turns out that twos' complement math works in all cases, with the carry being ignored.

Now that you've mastered that, I'll let you off the hook a bit by saying that none of this knowledge will be specifically required in any programs in this particular book. It is, however, a good thing to know about and is very useful in understanding the next idea, that of the sign and overflow flags in the Status Register.

The Sign Bit Flag

Since bit 7 of any byte can represent whether the number is positive or negative, a flag in the Status Register is provided for

easy testing of the nature of a given byte. Whenever a byte is loaded into a register, or when an arithmetic operation is done, the sign bit will be conditioned according to what the state of bit 7 is.

For example, LDA #\$80 will set the sign flag to one, whereas a LDA #\$40 would set it to zero. This is tested using the commands BPL and BMI. BPL stands for Branch on PLus, and BMI stands for Branch on MINus.

Regardless of whether you are using signed numbers or not, these instructions can be very useful for testing bit 7 of a byte. Many times bit 7 is used in various parts of the Apple to indicate the status of something. For example, the keyboard location, \$C000, gets the high bit set whenever a key is pressed.

Up until now we've always tested by comparing the value returned from \$C000 to \$80, such as in this listing:

```
1 *****
2 * KEYTEST PROG. 1 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8 *
9 KYBD EQU $C000
10 STROBE EQU $C010
11 *
12 CHECK LDA KYBD
13   CMP #$80
14   BCC CHECK ; NO KEYPRESS
15 *
16 CLR STA STROBE
17 END RTS
```

This program will stay in a loop until a key is pressed. The keypress is detected by the value returned from \$C000 being equal to or greater than \$80. A more elegant method is to use the BPL command:

```
1 *****
2 * KEYTEST PROG. 2 *
3 *****
4 *
5 *
6   OBJ $300
```

```

7   ORG $300
8   *
9   KYBD EQU $C000
10  STROBE EQU $C010
11  *
12  CHECK LDA KYBD
13   BPL CHECK ; NO KEYPRESS
14  *
15  CLR STA STROBE
16  END RTS

```

In this case, as long as the high bit stays clear (i.e. no key-press), the BPL will be taken and the loop continued. As soon as a key is pressed, bit 7 will be set to one, and the BPL will fail. The strobe is then cleared and the return done.

A similar technique is used for detecting whether or not a pushbutton has been pressed.

```

1  *****
2  *  BUTTON TEST*
3  *****
4  *
5  *
6  OBJ $300
7  ORG $300
8  *
9  PB0 EQU $C061
10 *
11 *
12 CHECK LDA PB0
13   BPL CHECK ; NO BUTTON PUSH
14 *
15 END RTS

```



CHAPTER 10

Disk Access

One of the more useful applications of machine language is in accessing the disk directly to store or retrieve data. You might do this to modify information already on the disk, such as when you're making custom modifications to DOS, or to deal with data within files on the disk, such as when you're patching or repairing damaged or improperly written files.

To cover DOS well requires more than a few chapters such as this. My intent here, then, is to supply you with enough information at least to access any portion of a disk and to have enough basic understanding of the overall layout of DOS and disks to make some sense of what you find there.¹

Here's what we'll cover in this chapter. First, we'll paint a general overview of what DOS is and how the data on the diskette is arranged. Then you'll learn a general access utility with which you can read and write any single block of data from a disk. With these, you'll have a starting point for your own explorations of this aspect of your Apple computer.

The Overview: DOS

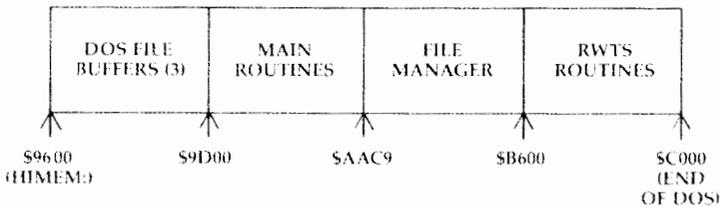
An Apple without a disk drive has no way of understanding commands like CATALOG or READ. These new words must enter

1. For a detailed look at DOS, I recommend the very recent book *Beneath Apple DOS*, by Dan Worth and Pieter Lechner (Reseda, CA: Quality Software, 1981).

its vocabulary from somewhere. When an Apple with a disk drive attached is first turned on or a PR#6 is done, this information is loaded into the computer by a process known as *booting*.

During the booting process, a small amount of machine language code on the disk interface card reads in data from a small portion of the disk. This data contains the necessary code to read another 10K of machine language referred to as DOS. This block of routines is responsible for all disk-related operations in the computer. It normally resides in the upper 10K or so of memory, from \$9600 to \$BFFF.

After booting, the organization of the memory used by DOS looks something like this:



The first area contains the three buffers set aside for the flow of data to and from the disk. A buffer is a block of memory reserved to hold data temporarily while it's being transferred. The MAXFILES command (a legal DOS command, see your manual if you haven't encountered it before), can alter the number of buffers reserved, and thus change the beginning address from \$9600 to other values. As it happens, three buffers are almost never needed, so, in a pinch for memory, you can usually set maxfiles to two, and often just one.

For example, if you had opened a text file called TEXTFILE, the data being read or written would be transferred via buffer one. If, while this file was still open, you did a catalog, buffer two would be put in use. If, instead, you opened two other files, say TEXTFILE 1 and TEXTFILE 2, and then tried to do a catalog, you would get a NO BUFFERS AVAILABLE error (assuming maxfiles was set at three). Buffer one starts at \$9AA6, buffer 2 at \$9853, and buffer 3 at \$9600. If maxfiles is set at three as in a normal system, it's occasionally useful to use the dead space of the unused buffer three for your own routines.

The main DOS routines starting at \$9D00 are the ones responsible for the interpreting commands such as CATALOG and in

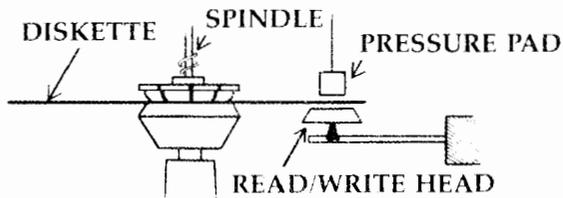
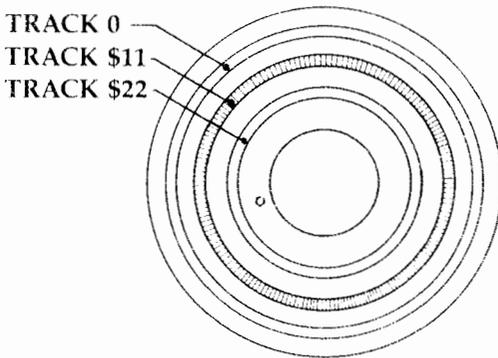
general, for allowing DOS to talk to Basic via control-D prefixed statements.

The file manager is a set of routines that actually executes the various commands sent via the main routines and that makes sure files are stored in a logical (well, almost) manner on the disk. This takes care of finding a file you name, checking to see if it's unlocked before a write, finding an empty space on the disk for new data, and countless other tasks required to store even the simplest file.

When the file manager gets ready to read data from or write data to the disk, it makes use of the remaining routines, called the RWTS routines. This stands for Read/Write Track Sector. To understand fully what this section does, though, it will be necessary now to look at the general organization of the disk itself.

Diskette Organization

Physically, a diskette is made of a material very similar to magnetic recording tape and is used by small portions of the surface being magnetized to store the required data in the form of ones and zeros.



But the diskette is more analogous to a record than to a continuous strip of tape. Arranged in concentric circles, there are thirty-five individual *tracks*, each of which is divided into sixteen segments called *sectors*.²

Tracks are numbered from 0 to 34 (\$00 to \$22), starting with Track 0 in the outer position and track 34 nearest the center. Sectors are numbered from 0 to 15 (\$00 to \$0F) and are interleaved for fastest access. This means that sector 1 does not immediately follow sector 0 when moving in the opposite direction from the diskette rotation. Rather, the order is:

0 - D - B - 9 - 7 - 5 - 3 - 1 - E - C - A - 8 - 6 - 4 - 2 - F

By the time DOS has read in and processed one sector, it doesn't have sufficient time to read the very next sector properly. If the sectors were arranged sequentially, DOS would have to wait for another entire revolution to read the next sector. By examining the sequence, you can see that after reading sector 0, DOS can let as many as six other sectors go by and still have time to start looking for sector 1. This alternation of sectors is sometimes called the *skew factor* or just *sector interleaving*.

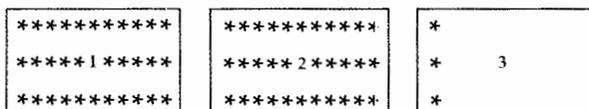
Looking for a given sector is done with two components. The first is a physical one, wherein the read/write head is positioned at a specific distance from the center to access a given track. The sector is located via software by looking for a specific pattern of identifying bytes. In addition to the 256 bytes of actual data within a sector, each sector is preceded by a group of identifying and error-checking bytes. These include, for example, something like \$00/03/FE for track \$00, sector \$03, volume \$FE. By continuously reading these identification bytes until a match with the desired values occurs, a given sector may be accessed.

This software method of sector location is usually called *soft-sectoring*, and it's somewhat unique to the Apple. Most other microcomputers use *hard-sectoring*. Hard-sectoring means that hardware locates the sector as well as the track. Search is done by indexing holes located around the center hole of a disk. Even Apple diskettes have this center hole, along with one to sixteen matching holes in the media itself, but these aren't actually used

2. Throughout this discussion, we will assume DOS 3.3, which uses sixteen sectors per track. DOS 3.2 has only thirteen sectors per track but is rapidly becoming obsolete. If you're using DOS 3.2, the correction from sixteen to thirteen should be made in the topics throughout.

by the disk drive. Because the Apple doesn't need these holes to index, using the second side of a disk is just a matter of notching the edge of the disk protector to create another write-protect tab. We'll not go into the pros and cons of using the second side but will leave that to you. It's one of those topics guaranteed to be worth twenty to thirty minutes of conversation at any gathering of two or more Apple owners.

Each sector holds 256 (100) bytes of data. This data must always be written or read as a single block. Large files are therefore always made up of multiples of 256 bytes. Thus a 520 byte file would take up three entire sectors, even though most of the third sector will be wasted space ($520 = 2 \times 256 + 8$)



Certain tracks and sectors are reserved for specific information. Track 17 (11) for example contains the directory. This gives each file a name, and also tells how to find out which sectors on the disk contain the data for each file. Track 17, Sector 0 contains the volume table of contents (VTOC), which is a master table of which sectors currently hold data, and which sectors are available for storing new data. If all of track 17 is damaged, it may be nearly impossible to retrieve any data from the disk even though the files themselves may still be intact.

The other main reserved area is on tracks 0 through 2. This is where the DOS that will be loaded when the disk is booted is held. If any of these tracks are damaged, it will not be possible to boot the diskette, or if the disk does boot, DOS may not function properly.

As a variation on this theme, by making certain controlled changes to DOS directly on the disk you can create your own custom version of DOS to enhance what Apple originally had in mind. These enhancements will become part of your system whenever you boot your modified diskette. Some modifications of this type are discussed below.

To gain access to a sector to make these changes, however, we need to be able to interface with the routines already in DOS to do our own operations. This is most easily done by using the RWTS routines mentioned earlier. Fortunately, Apple has made them fairly easy to use from the user's machine language program.

To use RWTS, you must do three general operations:

- 1) Specify the track and sector you wish access to.
- 2) Specify where the data is to be loaded to or read from (that is, give the buffer address).
- 3) Finally, call RWTS to do the read/write operation.

If the operation is to be a read, then we would presumably do something with the data in the buffer after the read is complete. If a write is to be done, then the buffer should be loaded before calling RWTS with the appropriate data. Usually, the way all this works is to read in a sector first, then make minor changes to the buffer, and then write the sector back out to the diskette.

Steps 1 and 2 are actually done in essentially the same operation, by setting up the IOB table (for Input/Output and control Block). This is described in detail (along with the sector organization) in the *Apple DOS Manual*, but here's enough information to *make you dangerous* as the saying goes.

The IOB table is a table you make up and place at a location of your choice. (You can also make use of the one already in memory that is used in DOS operations.) Most people I know seem to prefer to make up their own, but my personal preference is to use the one in DOS. Since most people I know aren't at this keyboard right now, I'll explain how to set up the table in DOS.

The table is made up of seventeen bytes and starts at \$B7E8. It's organized like this:

Location	Code	Purpose
\$B7E8	\$01	IOB type indicator, must be \$01.
B7E9	60	Slot number times sixteen (notice that this calculation, like multiplying by ten in decimal, means just moving the hex digit to the left one place).
B7EA	01	Drive number.
B7EB	00	Expected volume number.
B7EC	12	Track number.
B7ED	06	Sector number.
B7EE	FB	Low-order byte of device characteristic table.
B7EF	B7	High-order byte of D.C.T.
B7F0	00	Low-order byte of data buffer starting address.
B7F1	20	High-order byte of data buffer
B7F2	00	Unused.
B7F3	00	Unused.
B7F4	02	Command code; \$02 = write
B7F5	00	Error code (or last byte of buffer read in).
B7F6	00	Actual volume number
B7F7	60	Previous slot number accessed.
B7F8	01	Previous drive number accessed.

Because DOS has already set this table up for you, it isn't necessary to load every location with the appropriate values. In fact, if you're willing to continue using the last accessed disk drive, you need only specify the track and sector, set the command code, and then clear the error and volume values to 00. However, for complete accuracy, the slot and drive values should also be set so you know for sure what the entry conditions are.

Once the IOB table has been set up, the Accumulator and Y-Register must be loaded with the high and low-order bytes of the IOB table, and then the JSR to RWTS must be done. Although RWTS actually starts at \$B7B5, the call is usually done as JSR \$3D9 when it first boots. The advantage of calling here is that if Apple ever changes the location of RWTS, only the vector address at \$3D9 will be changed and a call to \$3D9 will still work.

A *vector* is the general term used for a location in memory that holds the information for a second address in memory. A vector is used so that a jump to the same place in memory can be routed to a number of other locations in memory, usually the beginning of various subroutines. A vector is rather like a telephone switchboard. Even though the user always calls the same address, the program flow can be directed to any number of different places simply by changing two bytes at the vector location.

For example, suppose at location \$3F5 we were to put these three bytes:

```
3F5: 4C 00 03
```

Listed from the Monitor, this would disassemble as:

```
03F5- 4C 00 03    JMP    $0300
```

Now whenever you do a call to \$3F5, either by a CALL 1013 or 3F5G, the program will end up calling a routine at \$300. It would now be a simple matter to write a switching program that would rewrite the two bytes at \$3F6 and \$3F7 so that a call to \$3F5 would go anywhere we wanted.

As it happens \$3F5 is used in just such a fashion by the ampersand (&) function of Applesoft. The *Applesoft II Basic Reference Manual* provides more information on this particular feature.

The best way to finish explaining how to use the IOB table

and RWTS is to present the following utility to access a given track and sector using RWTS. We'll then step through the program and learn why the various steps are done to use RWTS successfully.

```
1 *****
2 *
3 * GEN'L PURPOSE RWTS *
4 * DOS UTILITY
5 *
6 *****
7 *
8 *
9 OBJ $300
10 ORG $300
11 *
12 CTRK EQU $06
13 CSCT EQU $07
14 UDRIV EQU $08
15 USLOT EQU $09
16 BP EQU $0A ; BUFFER PTR.
17 UERR EQU $0C
18 UCMD EQU $E3
19 * USER SETS THIS TO HIS CMD
20 *
21 RWTS EQU $3D9
22 *
23 * BELOW ARE LOCS IN IOB
24 SLOT EQU $B7E9
25 DRIV EQU $B7EA
26 VOL EQU $B7EB
27 TRACK EQU $B7EC
28 SECTOR EQU $B7ED
29 BUFR EQU $B7F0
30 CMD EQU $B7F4
31 ERR EQU $B7F5
32 OSLOT EQU $B7F7
33 ODRIV EQU $B7F8
34 *
35 READ EQU $01
36 WRITE EQU $02
37 *
38 *
39 *
40 *****
41 * ENTRY CONDITIONS: SET *
42 * TRACK, SECTOR, SLOT, DR, *
43 * BUFFER AND COMMAND. *
44 *****
45 *
46 *
```

```

47 *
48 CLEAR LDA #$00
49   STA VOL
50 *
51   LDA USLOT
52   STA SLOT
53 *
54   LDA UDRIV
55   STA DRIV
56 *
57   LDA CTRK
58   STA TRACK
59 *
60   LDA CSCT
61   STA SECTOR
62 *
63   LDA UCMD
64   STA CMD
65 *
66   LDA BP
67   STA BUFR
68   LDA BP + 1
69   STA BUFR + 1
70 *
71   LDA #$B7
72   LDY #$E8
73   JSR RWTS
74   BCC EXIT
75 *
76 ERRHAND LDA ERR
77   STA UERR
78 *
79 EXIT RTS
80 *

```

This should list from the Monitor as:

```
*300L
```

```

0300- A9 00      LDA  #$00
0302- 8D EB B7   STA  $B7EB
0305- A5 09      LDA  $09
0307- 8D E9 B7   STA  $B7E9
030A- A5 08      LDA  $08
030C- 8D EA B7   STA  $B7EA
030F- A5 06      LDA  $06
0311- 8D EC B7   STA  $B7EC
0314- A5 07      LDA  $07

```

```

0316-   8D ED B7   STA   $B7ED
0319-   A5 E3     LDA   $E3
031B-   8D F4 B7   STA   $B7F4
031E-   A5 0A     LDA   $0A
0320-   8D F0 B7   STA   $B7F0
0323-   A5 0B     LDA   $0B
0325-   8D F1 B7   STA   $B7F1
0328-   A9 B7     LDA   #$B7
032A-   A0 E8     LDY   #$E8
032C-   20 D9 03   JSR   $03D9
032F-   90 05     BCC   $0336
0331-   AD F5 B7   LDA   $B7F5
0334-   85 0C     STA   $0C
0336-   60         RTS

```

When this program runs, it assumes the user has set the desired values for the track and sector wanted, which slot and drive to use, where the buffer is and whether to read or write.

Starting with the first functional line, line 48, the byte for the volume number in the IOB table (VOL) is stuffed with a zero. A value of zero here tells RWTS any volume number is acceptable during the access. If we wanted to access only a particular volume number, a value from \$01 to \$FF would be used instead of \$00.

In the next four sets of operations, the user values for the slot, drive, track, and sector numbers are put into the IOB table. Notice that, to have this work properly, you must set USLOT (\$09) to sixteen times the value for the slot you wish to use. For example, to access slot five you would store a \$50 (80 decimal) in location \$09 just before calling this routine.

The next pair of statements take the user command UCMD and put that in the table. If you want to read a sector, set UCMD = \$01. A write is UCMD = \$02. A few other options are seldom used. These are described in more detail in the DOS 3.3 manual in the section on RWTS.

Next, the buffer pointer is set to the value given by the user locations \$0A and \$0B. The required space is 256 bytes (\$100) and can be put anywhere that this won't conflict with data already in the computer. Convenient places are the number three DOS file buffer (\$9600), the input buffer itself (\$200), or an area of memory below \$9600 protected by setting HIMEM to an appropriate value.³ In the examples that follow, I'll use the area from

3. Note: The input buffer can only be used temporarily during your own routine. If you return to Basic, or do any input or DOS commands, data in this area will be destroyed. Other than that, it's a handy place to use.

\$1000 to \$10FF because no Basic program will be running and it's a nice number. In this case, \$0A and \$0B will be loaded with \$00 and \$10, respectively.

Last of all, the Accumulator is loaded with #\$B7 and the Y-Register with #\$E8, the high-order and low-order bytes of the IOB table address.

After the call to RWTS via the vector at \$3D9, the carry flag is checked for an error. If the carry is clear, there was no error and the routine returns via the RTS. If an error is encountered, the code will be transferred from the IOB table to the user location. The possible error codes are:

CODE	CONDITION
------	-----------

- | | |
|------|---|
| \$10 | Disk write-protected, and cannot be written to. |
| \$20 | Volume mismatch error. Volume number found was different than specified. |
| \$40 | Drive error. An error other than the three described here is happening (I/O error, for example). |
| \$80 | Read error RWTS will try forty-eight times to get to a good read; if it still fails, it will return with this error code. |

DOS Modifications

The ERR byte of the IOB table is somewhat unusual in that it does not remain at zero, even if the read/write operation was successful. In actual operation, if an error does not occur, the ERR byte will contain the last byte of the sector just accessed.

It is important therefore to always use the carry flag to detect whether an error has occurred or not. In fact, as your experience grows, you will notice that a great many subroutines use the carry flag as an indicator of the results of the operation. In the case of RWTS, the carry will be cleared if the access was successful, and set if an error occurred. It is not necessary to condition the carry before calling RWTS.

One of the best ways to grasp this routine is to use it to modify the DOS on a sample disk and observe the differences. Before proceeding with the examples, boot on an Apple master disk, then INIT a blank disk. This will be our test piece so to speak. Do not try these experiments on a disk already containing important data. If done correctly, the changes wouldn't hurt, but if an error were to occur you could lose a good deal of work!

#1: Disk-Volume Modification. First install the sector access

routine at \$300. Now insert the sample diskette. Enter the Monitor with CALL - 151 and type in:

```
*06: 02 02 01 60 00 10
*E3: 01
```

This assumes your diskette is in drive one, slot six. Now enter:

```
*300G
```

The disk motor should come on. When it stops type in:

```
*10AFL
```

You should get something like this:

```
10AF- A0 C5 LDY   #C5
10B1- CD D5 CC   CMP   $CCD5
10B4- CF                ???
10B5- D6 A0        DEC   $A0,X
10B7- CB                ???
10B8- D3                ???
10B9- C9 C4        CMP   #C4
```

This apparent nonsense is the ASCII data for the words *disk volume*, in reverse order. This is loaded in when the disk is booted and is used in all subsequent catalog operations.

The data was retrieved from track 2, sector 2, and put in a buffer starting at \$1000. The sequence we're interested in starts at byte \$AF in that sector. To modify that, type in:

```
*10AF: A0 D4 D3 C5 D4 A0 AD
*E3: 02
*300G
```

The first line rewrites the ASCII data there, the E3:02 changes the command to "write," and the 300G puts it back on the disk.

Now reboot on the disk and then type in CATALOG. When the catalog prints to the screen, the new characters "DISKTEST 254" should appear. By using an ASCII character chart, you can modify this part of the diskette to say anything you wish within the twelve-character limit.

#2: *Catalog Keypress Modification*. Reinstall the sector access

utiltiy, put the sample disk in the drive again, and type in:

```
*06: 01 0D 01 60 00 10
*E3: 01
*300G
```

This will read track 1, sector \$0D, into the buffer. Type in:

```
*1039L
```

The first line listed should be:

```
1039- 20 0C FD JSR $FD0C
```

Change this to:

```
*1039: 4C DF BC (JMP $BCDF)
```

And rewrite to the disk:

```
*E3: 02
```

Now read in the section corresponding to \$BCDF (track 0, sector 6) by typing:

```
*06 00 06
*E3: 01
*300G
```

And alter this section with:

```
*10DF: 20 0C FD C9 8D D0 03 4C 2C AE 4C 3C AE
*E3: 02
*300G
```

As it happens, this part of the disk isn't used and provides a nice place to put this new modification.

When you reboot after making this change, place a disk with a long catalog on it in the drive and type in `CATALOG`. When the listing pauses after the first group of names, press `RETURN`. The listing should stop, leaving the names just shown on the screen. If instead of pressing `RETURN` you press any other key, the catalog

will continue just as it normally would, going on to the next group of names.

Both these modifications will go into effect whenever you boot on the sample disk. These features can also be propagated to other disks by booting on the sample disk and using the new DOS to init fresh disks.

Many modifications to the existing DOS can be made this way, and we haven't even started to talk about storing binary data in general.

Bell Modification and Drive Access

(1) The first time you call the access utility from the Monitor, it will return with just the asterisk prompt. After that, unless you hit reset or do a catalog, it will return with the asterisk and a beep. This is because the status storage byte for the Monitor (\$48) gets set to a nonzero value by RWTS. If the beep annoys you, modify the access utility to set \$48 back to #\$00 before returning.

(2) If you set the slot/drive values to something other than your active drive, the active drive will still be the one accessed when you do, for example, the next catalog. This is because DOS doesn't actually look at the last-slot/drive-accessed values when doing a catalog. Instead, it looks at \$AA66 for the volume number (usually #\$00), at \$AA68 for the drive number, and at \$AA6A for the slot number (times sixteen). If you have Basic or machine language programs on which you wish to change the active drive values without having to do a catalog or give another command, then just POKE or STA the desired values in these three locations. Have fun!



CHAPTER 11

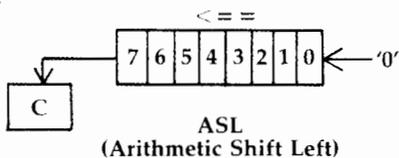
Shift Operators

Here I'd like to cover two main groups of machine language commands: *shift operators* and *logical operators*. Shifts are the easiest to understand, so we'll do them first.

You'll recall that the Accumulator holds a single eight-bit value, and that in previous programs it has been possible to test individual bits by examining flags in the Status Register. An example of this was used in testing bit 7 after an LDA operation. If the Accumulator is loaded with a value from \$00 to \$7F, bit 7 is clear and only BPL tests will succeed, since the sign flag remains clear. If, however, a value from \$80 to \$FF is loaded, BMI will succeed since bit 7 would be set; hence the sign flag will also be a one.

The shift commands greatly extend our ability to test individual bits by giving us the option of shifting each bit in the

Accumulator one position to the left or right. There are two direct shift commands, ASL (Arithmetic Shift Left) and LSR (Logical Shift Right).



In the case of ASL, each bit is moved to the left one position, with bit 7 going into the carry and bit 0 being forced to zero. In addition to the carry, the sign and zero flags are also affected. Some examples appear in the following listing.

ASL

Value		Result	(C)	(N)	(Z)
Hex	Binary	Hex	Carry	Sign	Zero
\$00	0 0 0 0 0 0 0 0	\$00	0	0	1
\$01	0 0 0 0 0 0 0 1	\$02	0	0	0
\$80	1 0 0 0 0 0 0 0	\$00	1	0	1
\$81	1 0 0 0 0 0 0 1	\$02	1	0	0
\$FF	1 1 1 1 1 1 1 1	\$FE	1	1	0

In the first case, there's no net change to the Accumulator, although the carry and sign flags are cleared and the zero flag is set. The 0 at each bit position was replaced by a 0 to its right.

However, in the case of \$01, the value in the Accumulator doubles to become \$02 as the 1 in bit 0 moves to the bit 1 position. In this case, all three flags will be cleared.

When the starting value is \$80 or greater, the carry will be set. In the case of \$80 itself, the Accumulator returns to zero after the shift, since the only 1 in the pattern, bit 7, is pushed out into the carry.

Notice that in the case of \$FF, the sign flag gets set as bit 6 in the Accumulator, and moves into position 7. Remember that in some schemes, bit 7 is used to indicate a negative number.

The ASL has the effect of doubling the byte being operated on. This can be used as an easy way to multiply by two. In fact, by using multiple ASLs, you can multiply by two, four, eight, sixteen, and so on, depending on how many you use. In the discussion of DOS and RWTS in chapter ten, you might remember that the IOB table required the slot number byte in the table to be sixteen times the true value. If you didn't want to do the multiplication ahead of time, you could do it in your access program, as below.

```

      .
      .
      .
A5 09      LDA  USLOT
0A         ASL
0A         ASL
0A         ASL
0A         ASL
8D E9 B7   STA  SLOT
      .
      .
      .

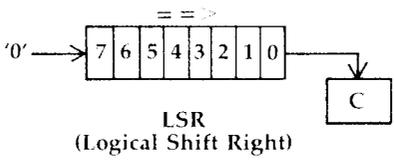
```

USLOT holds the value from one to seven that you pass to the routine and SLOT is the location in the IOB table in which the value for USLOT *16 should be placed. Even though the four ASLs look a bit redundant, notice that they only took four bytes. Actually, the LDA/STA steps consumed more bytes (five) than the four ASLs.

In general then, ASL is used for these types of operations:

- (1) Multiply by two, four, eight, and so on.
- (2) Set or clear the carry *for free* while shifting for some other reason.
- (3) Test bits 0 through 6. *Note:* This can be done, but it's usually only done this way for bit 6; there are, in general, better ways of testing specific bits, which we'll describe shortly.

The complement of the ASL command is the LSR. It behaves identically except that the bits all shift to the right.



The LSR can be used to *divide* by multiples of two. It's also a nice way to test whether a number is even or odd. Even numbers always have bit 0 clear. Odd always have it set. By doing an LSR followed by BCC or BCS, you can test for this. Whether a number is odd or even is sometimes called its *parity*. An even number has a parity of zero, and an odd number, a parity of one.

LSR also conditions the sign and zero flags.

In both LSR and ASL, one end or the other always gets forced to a zero. Sometimes this is not desirable. The solution to this is the *rotate* commands, ROL and ROR (ROtate Left, ROtate Right).

is to print characters on the screen *only* when both the character *and* the keyboard buffer show bit 7 set to one.

We can draw a simple chart that illustrates all the possibilities (and you know how fond computer people are of charts).

	Character Bit 7:	
	0	1
Keyboard Bit 7:	0	0
	1	1

The chart shows four possibilities. If the character's bit 7 is zero (a non-standard character) and the keyboard bit is zero (no keypress), then don't print the character (a zero result). Likewise, if only one of the conditions is being met but not the other, then the result is still zero, or don't print. Only when both desired conditions exist will we be allowed to print, as shown by the one as the result.

Taken to its extreme, what we end up with is a new mathematical function, AND. In the case of a single binary digit (or perhaps we should call it a *bigit*), the possibilities are few, and the answers are given as a simple zero or one.

What about larger numbers? Does the term 5 AND 3 have meaning? It turns out that it does, although the answer in this case will not be 8, and it is now that we must be cautious not to let our daily use of the word *addition* be confused with our new meaning.

As we look at these numbers on a binary level, how to get the result of 5 AND 3 will be more obvious.

$$\begin{array}{rcl}
 x & = & 5 \qquad \quad 0 \ 1 \ 0 \ 1 \\
 y & = & 3 \qquad \quad 0 \ 0 \ 1 \ 1 \\
 \hline
 x \text{ AND } y & & 0 \ 0 \ 0 \ 1 = 1
 \end{array}$$

If we take the chart created earlier and apply it to each set of matching bits in x and y , we can obtain the result shown. Starting on the left, two 0's give 0 as a result. For the next two bits, only a single 1 is present, in each case, still giving 0 as a result. Only in the last position do we get the necessary 1's in bit 0 of *both* numbers to yield a 1 in the result.

Thus 5 AND 3 does have meaning, and the answer is 1. (Try that at parties!)

Don't be discouraged if you don't see the immediate value in this operation; you should guess by now that everything is good for something!

AND is used for a variety of purposes. These include:

- (1) To force zeros in certain bit positions.
- (2) As a mask to let only ones in certain positions through.

When an AND operation is done, the contents of the Accumulator are ANDed with another specified value. The result of this operation is then put back in the Accumulator. The other value may be either given by way of the immediate mode or held in a memory location. These are some possible ways of using AND:

```
LDA  #$80
AND  #$7F
AND  $06
AND  $300,X
AND  ($06),Y
```

To understand better how the AND is used, we should clarify some other ideas. One of these is the nature of machine language programs in general. I believe that, at any given point in a program, one of two kinds of work will be going on. One is the operational mode, where some specific task, such as clicking a speaker or reading a paddle, is taking place. At these moments, data as such does not exist. In the other case, the processing mode, data has been obtained from an operational mode, and the information is processed and/or passed to some other routine or location in memory.

A given routine is rarely made up of just one mode or the other, but any given step usually falls more into one category than the other.

These ideas are important because, in general, all the logical operators are used during the processing phases of a program. At those times, some kind of data is being carried along in a register or memory location. Part of the processing that occurs is often done with the logical operators.

In the case of the two modes of use, operational and processing, we are really just talking about two different ways of looking at the same operation. To illustrate this, examine this partial disassembly of the Monitor starting at \$FDED:

*FDEDL

FDED-	6C	36	00	JMP	(\$0036)
FDF0-	C9	A0		CMP	#\$A0
FDF2-	90	02		BCC	\$FDF6
FDF4-	25	32		AND	\$32
FDF6-	84	35		STY	\$35
FDF8-	48			PHA	
FDF9-	20	78	FB	JSR	\$FB78
FDFC-	68			PLA	
FDFD-	A4			LDY	\$35
FDFE-	60			RTS	

For normal text output on the Apple, the Accumulator is loaded with the ASCII value for the character to be printed, the high bit is set, and a JMP to COUT (\$FDED) is done. From looking at the listing, you can see that at \$FDED is an indirect jump based on the contents of \$36,37 (called a vector).

If this seems a little vague, then consider for a moment what I call the *flow of control* in the computer. This means that the computer is *always* executing a program somewhere. Even when there's nothing but a flashing cursor on the screen, the computer is still in a loop programmed to get a character from the keyboard. When you call your own routines, the computer is just temporarily leaving its own activities to do your tasks until it hits that last RTS. It then goes back to what it was doing before; usually, that's waiting for your next command.

When characters are printed to the screen, disk, printer, or anywhere else, there's a flow of control that carries along the character to be printed. For virtually *every* character printed, the 6502 scans through this region as it executes the necessary code to print the character.

Normally, \$36,37 points to \$FDF0 (at least before DOS is booted). This may seem a little absurd until you realize that a great deal of flexibility is created by the vector. For instance, a PR#1, such as you do when turning on a printer, redirects \$36,37 to point to the card, which in turn, after printing a character, returns usually to where \$36,37 *used* to print.

The card thus borrows the flow of control long enough to print the character, after which it gives control back to the screen print routine. Likewise, when DOS is booted, \$36,37 gets redirected from \$FDF0 to \$9EBD, which is where phrases preceded by a control-D are detected. If no control-D is found, the output is returned to \$FDF0.

Now, back to what the AND is used for. Normally when the routine enters at \$FDF0, the Accumulator will hold a value between \$80 and \$DF. The characters from \$80 to \$9F are all control characters and are passed through by the BCC following the first CMP. Characters passing this test will be the usual alphabetic, numeric, and special characters shown on the Screen Character Chart in appendix E. You'll notice at this point an AND with the contents of \$32 is done. Location \$32 is called INVFLG and usually holds either \$FE, \$7E, or \$3F depending on whether the computer is in the NORMAL, FLASHING, or INVERSE text mode. Let's assume that the Accumulator is holding the value for a normal A. Look at the following table to see what happens when an AND is done with each of these values.

EXAMPLE 1:	HEX	BINARY	ASCII
Accumulator:	\$C1	1 1 0 0 0 0 0 1	A
INVFLG:	\$FF	1 1 1 1 1 1 1 1	—
Result:	\$C1	1 1 0 0 0 0 0 1	A

EXAMPLE 2:	HEX	BINARY	ASCII
Accumulator:	\$C1	1 1 0 0 0 0 0 1	A
INVFLG:	\$7F	0 1 1 1 1 1 1 1	—
Result:	\$41	0 1 0 0 0 0 0 1	A (flashing)

EXAMPLE 3:	HEX	BINARY	ASCII
Accumulator:	\$C1	1 1 0 0 0 0 0 1	A
INVFLG:	\$3F	0 0 1 1 1 1 1 1	—
Result:	\$01	0 0 0 0 0 0 0 1	A (inverse)

In the first example, ANDing with \$FF yields a result identical to the original value. The result is identical because, with each bit set to one, the resulting bit will always come out the same as the corresponding bit in the Accumulator. (Can you guess what the result of ANDing with \$00 would always yield?) This means that the character comes out in its original form.

In the second case, ANDing with \$7F has the effect of forcing a zero in bit 7 of the result. Examining the Screen Character Chart in the appendix, we can see that \$41 corresponds to a flashing A.

The Apple uses the leading two bits to determine how to print the character. If the leading two bits are *off*, then the character will be in inverse. If bit 7 is zero and 6 is one, then the character will be printed in flashing mode. If bit 7 is set, then the character will be displayed in normal text.

Using the AND operator forces a zero in the desired positions and lets the remaining bit pattern through.

In general, then, the way to use the AND is to set a memory location equal to a value wherein all bits are set to one except for those that you wish to force to zero.

You can also think of AND as acting rather like a screen that lets only certain parts of the image through. When INVLG is set to \$3F, the leading bits will always be zero, regardless of whether they were set at entry or not; hence, the expression *mask*.

Sometimes figuring exactly what value you should use for the desired result is tricky. As a general formula, first decide what bits you want to force to zero and then calculate the number with all other positions set to ones. This will give the proper value to use in the mask. For example, to derive the inverse display mask value:

- (1) Determine which bits to force to zero:

0 0 x x x x x

- (2) Calculate with the remaining positions set to ones:

0 0 1 1 1 1 1 1 = \$3F (63)

Try this with the desired result of forcing bit 7 only to zero and see if you get the proper value for INVLG of \$7F.

BIT

The command somewhat related to the AND is BIT. This is provided to allow the user to determine easily the status of specific bits. When BIT is executed, quite a number of things happen. First of all, bits 6 and 7 of the memory location are transferred directly to the sign and overflow bits of the Status Register. Since we've not discussed the overflow flag, let me say briefly that its related commands, BVC and BVS, may be used just as BPL and BMI are used to test the status of the sign flag. If V (the overflow flag) is clear, BVC will succeed. If V is set, BVS will work.

Most important, though, is the conditioning of the zero flag. If one or more bits in the memory location match bits set in the Accumulator, the zero flag will be cleared ($Z = 0$). If no match is made, Z will be set ($Z = 1$). This is done by ANDing the Accumulator and the memory location and conditioning Z

appropriately. The confusing part is that this may seem somewhat backward. Alas, it's unavoidable; it's just one of those notes to scribble in your book so as to remember the quirk each time you use it.

Note that one of the main advantages of BIT is that the Accumulator is unaffected by the test.

Here are examples of how BIT might be used:

EXAMPLE 1: To test for bits 0 and 3, set:

```
LDA    #$05    ; 0000 0101
BIT    MEM
BNE    OK      ; (1 OR MORE BITS MATCH)
```

EXAMPLE 2: To test for bit 7, set in memory:

```
CHK   BIT     $C000    ; (KEYBOARD).
      BPL     CHK      ; (BIT 7 CLR, NO KEY PRESSED)
      BIT     $C010    ; (ACCESS $C010 TO CLR STROBE) . . .
```

If you want to test for *all* of a specific set of bits being on, the AND command must be used directly.

EXAMPLE 3: To test for both bits 6 and 7 being on:

```
LDA    CHAR    ; '1100 0000'
AND    #$C0
CMP    #$C0
BEQ    MATCH   ; BOTH BITS "ON"
```

This last example is somewhat subtle, in that the result in the Accumulator will only equal the value with which it was ANDed if each bit set to one in the test value has an equivalent bit on the Accumulator.

ORA and EOR

These last two commands bring up an interesting error of sorts in the English language, and that is the difference between an *inclusive* OR and the *exclusive* OR. What all this is about is the phenomenon that saying something like "I'll go to the store if it stops raining or if a bus comes by" has two possible interpretations. The first is that if *either* event happens, and even if both

events occur, then the result will happen. This is called an *inclusive* OR statement.

The other possibility is that the conditions to be met must be one or the other but not both. This might be called the purest form of an OR statement. It is either night or day, but never both. This would be called an *exclusive* OR statement.

In machine language, the inclusive OR function is called ORA or OR Accumulator. The other is called EOR for Exclusive OR. The figure below shows the charts for both functions.

		(Acc)				(Acc)	
		0	1			0	1
(Memory)	0	0	1			0	1
	1	1	1			1	0

First, consider the table for ORA. If either or both corresponding bits in the Accumulator and the test value match, then the result will be a one. Only when neither bit is one does a zero value for that bit result. The main use for ORA is to force a one at a given bit position. In this manner, it's something of the complement to the use of the AND operator to force zeros.

The following table presents some examples of the effect of the ORA command.

	Example 1:	Example 2
Accumulator:	\$80 1000 0000	\$83 1000 0011
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$83 1000 0011	\$8B 1000 1011

Use of ORA conditions the sign and zero flags, depending on the result, which is automatically put into the Accumulator.

The EOR command is somewhat different in that the bits in the result are set to one only if one or the other of the corresponding bits in the Accumulator and test value is set to one, but not both.

EOR has a number of uses. The most common is in encoding data. An interesting effect of the table is that, for any given test value, the Accumulator will flip back and forth between the original value and the result each time the EOR is done. See the examples in the table below.

Accumulator:	\$80 1000 0000	\$83 1000 0011
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$83 1000 0011	\$89 1000 1001

Accumulator:	\$83 1000 0011	\$89 1000 1001
Value:	\$03 0000 0011	\$0A 0000 1010
Result:	\$80 1000 0000	\$83 1000 0010

This flipping phenomenon is used extensively in hi-res graphics to allow one image to overlay another without destroying the image below. EOR can also be used to reverse specific bits. Simply place ones in the positions you wish reversed.

You might find it quite rewarding to write your own experimental routine that will EOR certain ranges of memory with given values. Then make the second pass to verify that the data has been restored. This is especially interesting when done either on the hi-res screen or blocks of ASCII data such as on the text screen.

It would be a shame if you've stayed with this chapter long enough to read through all this and didn't get a program for your efforts, so I offer the demonstration program that follows. It provides a way of visually experimenting with the different shifts and logical operators. Assemble the machine language program listed and save it to disk under the name OPERATOR.OBJ.

```

1 *****
2 * BINARY FUNCTION DISPLAY *
3 *           UTILITY           *
4 *****
5 *
6 *
7           OBJ   $300
8           ORG   $300
9 *
10 NUM      EQU   $06
11 MEM      EQU   $07
12 RSLT     EQU   $08
13 STAT     EQU   $09
14 *
15 YSAV1    EQU   $35
16 COUT1    EQU   $FDF0
17 CVID     EQU   $FDF9
18 COUT     EQU   $FDED
19 PRBYTE   EQU   $FDDA
20 *
21 *

```

0300:	A9 00	22	OPERATOR	LDA	#\$00	
0302:	48	23		PHA		
0303:	28	24		PLP		
0304:	A5 06	25		LDA	NUM	
0306:	25 07	26		AND	MEM	; <= ALTER THIS
0308:	85 08	27		STA	RSLT	
030A:	08	28		PHP		
030B:	68	29		PLA		
030C:	85 09	30		STA	STAT	
030E:	60	31		RTS		
		32	*			
030F:	A9 A4	33	PRHEX	LDA	#\$A4	;'\$'
0311:	20 ED FD	34		JSR	COUT	
0314:	A4 06	35		LDA	NUM	
0316:	4C DA FD	36		JMP	PRBYTE	
		37	*			
0319:	A5 06	38	PRBIT	LDA	NUM	
031B:	A2 08	39		LDX	#\$0 8	
031D:	0A	40	TEST	ASL		
031E:	90 0D	41		BCC	PZ	
0320:	48	42	P0	PHA		
0321:	A9 B1	43		LDA	#\$B1	;'1'
0323:	20 ED FD	44		JSR	COUT	
0326:	A9 A0	45		LDA	#\$A0	;'SPC'
0328:	20 ED FD	46		JSR	COUT	
032B:	B0 0B	47		BCS	NXT	
		48	*			
032D:	48	49	PZ	PHA		
032E:	A9 B0	50		LDA	#\$B0	;'0'
0330:	20 ED FD	51		JSR	COUT	
0333:	A9 A0	52		LDA	#\$A0	;'SPC'
0335:	20 ED FD	53		JSR	COUT	
		54	*			
0338:	68	55	NXT	PLA		
0339:	CA	56		DEX		
033A:	D0 E1	57		BNE	TEST	
		58	*			
033C:	60	59	EXIT	RTS		
		60	*			
033D:	EA	61		NOP		
033E:	EA	62		NOP		
033F:	EA	63		NOP		
		64	*			
0340:	C9 80	65	CSHOW	CMP	#\$80	;STAND CHAR?
0342:	90 10	66		BCC	CONT	
0344:	C9 8D	67		CMP	#\$8D	; < C/R >
0346:	F0 0C	68		BEQ	CONT	
0348:	C9 A0	69		CMP	#\$A0	;'SPC'
034A:	B0 08	70		BCS	CONT	
		71	*			

```

034C:  48          72          PHA
034D:  84 35        73          STY  YSAV1
034F:  29 7F        74          AND  #$7F      ; FORCE '0' IN BIT 7
0351:  4C F9 FD     75          JMP  CVID
                                76 *
0354:  4C F0 FD     77 CONT     JMP  COUT1
                                78 *
0357:  00          79 EOF     BRK
                                80 *
                                81 *

```

Then enter the accompanying Apple program (Logical Operator Demo Program) and save it under the name OPERATOR DEMO PROGRAM.¹

Logical Operator Demo Program

```

0  IF PEEK (768) <> 169 THEN PRINT CHR$ (4);"BLOAD
  OPERATOR.OBJ,A$300"
5  POKE 54,64: POKE 55,3: CALL 1002: REM HOOK UP CTRL
  SHOW
10 REM LOGICAL OPERATOR PROG.
15 OP = 744:F = 768:PH = 783:PB = 793
20 TEXT : HOME : GOTO 1000
100 KEY = PEEK (-16384): IF KEY > 127 THEN 1000
110 A = PDL (0):A = PDL (0)
120 M = PDL (1):M = PDL (1)
125 POKE 6,A: POKE 7,M
130 CALL F: REM EVALUATE FUNCTION
140 R = PEEK (8):S = PEEK (9)
200 VTAB 11: HTAB 1: PRINT "OPCODE:": POKE 6,O: GOSUB
  500: VTAB 11: HTAB 32: PRINT """;O$;"
210 VTAB 14: PRINT "ACC:": POKE 6,A: GOSUB 500: HTAB 30:
  PRINT " "; HTAB 30: PRINT CHR$ (A): VTAB 14: HTAB 33: PRINT
  "(P0)": POKE 1742,A: IF A = 13 OR A = 141 THEN VTAB 14:
  HTAB 30: INVERSE : PRINT "M": NORMAL
215 IF O1 = 7 THEN VTAB 16: PRINT "MEMORY:": POKE 6,M:
  GOSUB 500: HTAB 30: PRINT " "; HTAB 30: PRINT CHR$ (M):
  VTAB 16: HTAB 33: PRINT "(PI)": POKE 1998,M: IF M = 13 OR M
  = 141 THEN VTAB 16: HTAB 30: INVERSE: PRINT "M": NORMAL
220 IF O$ <> "BIT" THEN VTAB 18: PRINT "RESULT:": POKE
  6,R: GOSUB 500: HTAB 30: PRINT " "; HTAB 30: PRINT CHR$ (R):
  POKE 1270,R: IF R = 13 OR R = 141 THEN VTAB 18: HTAB 30:
  INVERSE: PRINT "M": NORMAL
230 VTAB 20: PRINT "STATUS:": POKE 6,S: GOSUB 500: PRINT
240 VTAB 22: HTAB 10: PRINT "N b V b --- b b b D b I b Z b C"
250 GOTO 100
499 END

```

1. In the PRINT statements of the following program, the symbol *b* represents a space or *blank*.

```

5000 REM PRINT BITS & HEX
510 HTAB 10: CALL PB: HTAB 26: CALL PH: RETURN
1000 REM SELECT FUNCTION
1010 T = PEEK ( - 16368):FC = FC + 1 - (KEY = 136)* 2:
IF FC > 8 THEN FC = 1
1011 IF KEY = 193 THEN FC = 1
1012 IF KEY = 194 THEN FC = 3
1013 IF KEY = 197 THEN FC = 4
1014 IF KEY = 204 THEN FC = 5
1015 IF KEY = 207 THEN FC = 6
1016 IF KEY = 210 THEN FC = 7
1019 IF FC < 1 THEN FC = 8
1020 FOR I = 1 TO FC: READ OS,O,O1: NEXT I: RESTORE
1025 IF KEY = 155 THEN PRINT CHR$ (4);"PR#0": END
1030 POKE OP,O: POKE OP + 1,O1: HOME
1050 ON FC GOSUB 1100,1200,1300,1400,1500,1600,1700,1800
1055 POKE 32,0
1060 A = -1: GOTO 100
1100 REM 'AND'
1110 POKE 32,9
1140 VTAB 1: PRINT "-----"
1145 PRINT "! B AND B ! B B O B B ! B B 1 B B !"
1150 PRINT "-----"
1155 PRINT "! B B O B B ! B B O B B ! B B O B B !"
1160 PRINT "-----"
1165 PRINT "! B B 1 B B ! B B O B B ! B B 1 B B !"
1170 PRINT "-----"
1175 PRINT : HTAB 7: PRINT "'AND'"
1180 VTAB 23: PRINT "'^ B B B B B B B B B B ^ B ^'"
1185 RETURN
1200 REM 'ASL'
1220 VTAB 1: HTAB 9: PRINT "----- < == -----"
1225 HTAB 4: PRINT "-----!7!6!5!4!3!2!1!0!< -- B '0'"
1230 HTAB 4: PRINT "! B B B B -----"
1235 HTAB 3: PRINT "----"
1240 HTAB 3: PRINT "'!C!'"
1245 HTAB 3: PRINT "----"
1250 VTAB 7: HTAB 16: PRINT "'ASL': HTAB 8: PRINT
"(ARITHMETIC SHIFT LEFT)"
1280 VTAB 23: HTAB 10: PRINT "^ B B B B B B B B B B ^ B ^"
1285 RETURN
1300 REM 'BIT'
1310 POKE 32,9
1340 VTAB 1: PRINT "-----"
1345 PRINT "! B AND B ! B B O B B ! B B 1 B B !"
1350 PRINT "-----"
1355 PRINT "! B B O B B ! B B O B B ! B B O B B !"
1360 PRINT "-----"
1365 PRINT "! B B 1 B B ! B B O B B ! B B 1 B B !"
1370 PRINT "-----"
1375 PRINT : HTAB 7: PRINT "'BIT'"
1380 VTAB 23: PRINT "M B M B B B B B B B B B ^": PRINT "7 B 6";
1385 RETURN
1400 REM 'EOR'
1410 POKE 32,9
1440 VTAB 1: PRINT "-----"

```

```

1445 PRINT "! B EOR B ! B B O B B ! B B 1 B B !"
1450 PRINT "-----"
1455 PRINT "! B B O B B ! B B O B B ! B B 1 B B !"
1460 PRINT "-----"
1465 PRINT "! B B 1 B B ! B B 1 B B ! B B O B B !"
1470 PRINT "-----"
1475 PRINT : HTAB 7: PRINT "'EOR'"
1480 VTAB 23: PRINT "^ B B B B B B B B B B B B ^"
1485 RETURN
1500 REM 'LSR'
1520 VTAB 1: HTAB 9: PRINT "----- = => -----"
1525 HTAB 2: PRINT "'0' B -- >!7!6!5!4!3!2!1!0! -----"
1530 VTAB 3: HTAB 9: PRINT "----- B B B B !"
1535 HTAB 29: PRINT "----"
1540 HTAB 29: PRINT "!C!"
1545 HTAB 29: PRINT "----"
1550 VTAB 7: HTAB 15: PRINT "'LSR'": HTAB 8: PRINT
"(LOGICAL SHIFT RIGHT)"
1580 VTAB 23: HTAB 10: PRINT "0 B B B B B B B B B B B B ^ B ^"
1585 RETURN
1600 REM 'ORA'
1610 POKE 32,9
1640 VTAB 1: PRINT "-----"
1645 PRINT "! B O R A B ! B B O B B ! B B 1 B B !"
1650 PRINT "-----"
1655 PRINT "! B B O B B ! B B O B B ! B B 1 B B !"
1660 PRINT "-----"
1665 PRINT "! B B 1 B B ! B B 1 B B ! B B 1 B B !"
1670 PRINT "-----"
1675 PRINT : HTAB 7: PRINT "'ORA'"
1680 VTAB 23: PRINT "^ B B B B B B B B B B B B ^"
1685 RETURN
1700 REM 'ROL'
1720 VTAB 1: HTAB 9: PRINT "-----"
1725 HTAB 4: PRINT "----- !7!6!5!4!3!2!1!0!< -----"
1730 HTAB 4: PRINT "! B B B B ----- B B B B !"
1735 HTAB 4: PRINT "! B B B B B B B B B B B B --- B B B B B B B B B B !"
1740 HTAB 4: PRINT "----- >!C! -----"
1745 HTAB 16: PRINT "----"
1750 VTAB 8: HTAB 15: PRINT "'ROL'": HTAB 9: PRINT
"(ROTATE ONE BIT LEFT)"
1780 VTAB 23: HTAB 10: PRINT "^ B B B B B B B B B B B B ^ B ^"
1785 RETURN
1800 REM 'ROR'
1820 VTAB 1: HTAB 9: PRINT "----- = => -----"
1825 HTAB 4: PRINT "----- > !7!6!5!4!3!2!1!0! -----"
1830 HTAB 4: PRINT "! B B B B ----- B B B B !"
1835 HTAB 4: PRINT "! B B B B B B B B B B B B --- B B B B B B B B B B !"
1840 HTAB 4: PRINT "----- !C!< -----"
1845 HTAB 16: PRINT "----"
1850 VTAB 8: HTAB 15: PRINT "'ROR'": HTAB 9: PRINT
"(ROTATE ONE BIT RIGHT)"
1880 VTAB 23: HTAB 10: PRINT "^ B B B B B B B B B B B B ^ B ^"
1885 RETURN

```

```
2000 DATA AND,37,7,ASL,10,234,BIT,36,7,EOR,69,7,LSR,  
74,234,ORA,5,7,ROL,42,234,ROR,106,234  
65535 REM COPYRIGHT (C) 1981  
65535 REM ROGER R. WAGNER
```

The basic theory of operation for the program is to rewrite locations \$306 and \$307 with the appropriate values to create the different operators. These values are contained in the DATA statement on line 2000 of the Applesoft program. In addition, there are routines to print the value in location \$06 in both binary and hex formats. Also, there is a routine to show control characters in inverse. You may wish to examine each of these to determine the logic, if any, behind their operation.

The Applesoft program itself operates by getting a value for the Accumulator and the memory location from paddles zero and one. The double read in lines 110 and 120 minimizes the interaction between the two paddles. Pressing any key advances the display to the next function; the left arrow backs up. Pressing A, B, E, L, O or R will jump to selected functions.

The screen display shows the hex and binary values for each number and also what character would be printed via a PRINT CHR\$(X) statement (control characters are shown in inverse). To the far right is the character obtained when the value is poked into the screen display part of memory.

I suppose if I were a purist, the entire thing would have been written in machine language. Oh well, maybe next time.

1000

9000

8000

7000

6000

5000

4000

3000

2000

1000

0

1000

2000

3000

4000



CHAPTER 12

I/O Routines: Print and Input

In chapter ten, I discussed how to access the disk using the RWTS routine. There is another way in which the disk can be read that is more similar to the procedure used in Basic. The advantage of this system is that we need not be concerned about what track and sector we're using, since DOS will handle the files just as it does in a *normal* program. The disadvantage is that we must have the equivalent of PRINT and INPUT statements to use in our programs to send and receive the data. So, before going any further, let's digress to input/output routines.

Print Routines

I, personally, have two favorite ways of simulating the PRINT statement. The first was described in earlier chapters and looks like this:

```
1 *****
2 * DATA-TYPE PRINT ROUTINE *
3 *****
4 *
5 *
6   OBJ $300
```

		7	ORG \$300
		8	*
		9	COUT EQU \$FDED
0300:	A2 00	10	*
		11	ENTRY LDX # \$00
0302:	BD 0E 03	12	LOOP LDA DATA,X
0305:	F0 06	13	BEQ DONE
0307:	20 ED FD	14	JSR COUT
030A:	E8	15	INX
030B:	D0 F5	16	BNE LOOP
		17	*(ALWAYS UP TO 255 CHRS)
		18	*
030D:	60	19	DONE RTS
		20	*
030E:	84	21	DATA HEX 84
030F:	C3 C1 D4	22	ASC "CATALOG"
0316:	8D 00	23	HEX 8D00
0318:	00	24	*
		25	EOF BRK

This type of routine uses a defined data block to hold the ASCII values for the characters we wish to print. The printing is accomplished by loading the X-Register with 00 and stepping through the data table until a 00 is encountered. Each byte loaded is put into the Accumulator and printed via the JSR to COUT (\$FDED). When the 00 is finally reached, the BEQ on line 13 is taken and we return from the routine via the RTS at DONE.

The new item of interest in this listing is the use of the \$84 as the first character printed. This will be printed as a control-D and the word CATALOG that follows will be executed as a DOS command.

The essence of this chapter's message, along with the routines, is that any DOS command can be executed from machine language exactly the same way it's done from Basic. One need only precede the command with a control-D and terminate the command with a carriage return. (READ and WRITE are something of an exception to this technique but can still be done with only minor adjustments.)

Because DOS looks at all characters being output, it will see the control-D character and behave accordingly.

Try entering this program and then calling with either a 300G from the Monitor, or a CALL 768 from Basic.

This next print routine is more involved but does offer some advantages. The advantage is that the hex or ASCII data for what you want to print can immediately follow the JSR PRINT state-

ment, which parallels Basic a little more closely, and avoids constructing the various data blocks. The disadvantage is that the overall code is longer for short programs such as this. The general rule of thumb is to use the data-type when you have only to print once or twice during the program, and to use the following type when printing many times.

The logic behind the operation of this second method is slightly more complex than the previous routine, but I think you'll find it quite interesting.

Here's the new method:

```

1 *****
2 * SPECIAL PRINT ROUTINE *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PTR EQU $46
9   COUT EQU $FDED
10 *
0300:  20 0D 03 11 ENTRY JSR PRINT
0303:  84      12 E0 HEX 84
0304:  C3 C1 D4 13   ASC "CATALOG"
030B:  8D 00   14   HEX 8D00
030D:  60      15 DONE RTS
      16 *
030E:  68      17 PRINT PLA
030F:  85 46   18   STA PTR
0311:  68      19   PLA
0312:  85 47   20   STA PTR +1
0314:  A0 01   21   LDY #$01 ; PTR HOLDS 'E0' - 1 HERE
      22 *
0316:  B1 46   23 P0 LDA (PTR),Y
0318:  F0 06   24   BEQ FNSH
031A:  20 ED FD 25   JSR COUT
031D:  C8      26   INY
031E:  D0 F6   27   BNE P0 ;(MOST ALWAYS)
      28 *
0320:  18      29 FNSH CLC
0321:  98      30   TYA
0322:  65 46   31   ADC PTR
0324:  85 46   32   STA PTR
0326:  A5 47   33   LDA PTR +1
0328:  69 00   34   ADC #$00
0329:  48      35   PHA
032B:  A5 46   36   LDA PTR
032D:  48      37   PHA
032E:  60      38 EXIT RTS

```

39 * WILL RTS TO DONE INSTEAD OF
40 *E0 !
41 *

This one is rather interesting in that it uses the stack to determine where to start reading the data. You'll recall that when a JSR is done, the return address minus one is put on the stack. Upon entry to the Print routine, we use this fact to put that address in PTR, PTR + 1. By loading the Y-Register with #E0 and indexing PTR to fetch the data, we can scan through the string to be printed until we encounter 00, which indicates the end of the string.

When the end is reached, the BEQ FNSH will be taken. After that happens, the Y-Register (the length of the string printed) is transferred to the Accumulator and added to the address in PTR, PTR + 1, and the result pushed back onto the stack. Remember that the old return address was E0 - 1 until it was pulled off.

Now when the RTS is encountered, the program will be fooled into returning to DONE instead of E0 as it would otherwise have done.

To summarize, then:

- 1) Any DOS command can be executed from machine language just as it is done in Basic by doing the equivalent of Printing a control-D followed by the command and a carriage return.
- 2) A data-type print routine uses ASCII characters in a labeled block, which is then called by name using the X-Register in a direct indexed addressing mode. The string to be printed should have the high bit set (ASCII value + \$80), and the string must be terminated by a zero (at least when using the routine given here).
- 3) A JSR to a special print routine can also be done. In this case the ASCII data should immediately follow the JSR. Again have the high bit set, and end with 00.

Input Routines

The other side of the coin is, of course, the Input routine. You might be surprised by the number of times I get calls from people saying, "If only the input in such-and-such program would accept quotes, commas, etc." The solution is actually quite simple and is presented here.

In its simplest form, the routine looks like this:

```

1 *****
2 * INPUT ROUTINE FOR BINARY *
3 *****
4 *
5 * STORES STRING AT PTR LOC
6 *
7   OBJ $300
8   ORG $300
9 *
10  GETLN EQU $FD6F
11  BUFF EQU $200
12  PTR EQU $46
13 *
14 *
15  ENTRY LDX #$00
16    JSR GETLN
17 *
18  CLEAR TXA
19    TAY ; Y-REG = LEN NOW
20    INY
21    LDA #$00
22    STA (PTR),Y; PUT END-OF-STRING MARKER
23  C2 LDA BUFF,Y
24    AND #$7F
25    STA (PTR),Y
26    DEY
27    CPY #$FF
28    BNE C2
29 *
30  DONE RTS
0300:  A2 00
0302:  20 6F FD
0305:  8A
0306:  A8
0307:  C8
0308:  A9 00
030A:  91 46
030C:  B9 00 02
030F:  29 7F
0311:  91 46
0313:  88
0314:  C0 FF
0316:  D0 F4
0318:  60

```

The heart of this routine is a call to the Monitor's GETLN routine, which gets a line of text from the keyboard or current input device and puts it in the keyboard buffer (\$200-2FF).

This saves our having to write a routine ourselves. The beauty of this method is also that all the escape and left/right arrow keys are recognized. When the routine returns from GETLN, the entered line is sitting at \$200+. The length is held in the X-Register.

At this point we could, presumably, just return from our routine as well, but as it happens, all the data now in the buffer has the high bit set—that is, #\$80 has been added to the ASCII value of each character. Because Applesoft in particular, and many other routines in general, don't expect this, the high bit should be cleared before returning. Also \$200+ will hold only one string at a time, so there should be some provision for putting the string to some final destination.

Both are accomplished in the Clear section of this routine.

First the length of the string is transferred via the TXA, TAY to the Y-Register. My preference is then to mark the end of the string by bumping Y by one and storing 00 as a terminator. That step is optional.

Next, C2 begins a loop that loads each character into the buffer, does an AND with #\$7F, and then stores the result at a location pointed to by PTR, PTR + 1 plus the Y-Register offset.

The AND #\$7F has the effect of clearing the high bit by forcing bit 7 to zero.

The Y-Register is then decremented and the loop repeated until the DEY forces Y to an \$FF. This will indicate that the last value was \$00, and we have thus completed scanning the buffer.

This routine will work fine as long as you're willing to manage the string entirely yourself once it gets to the PTR, PTR + 1 location.

As noble as it might be to write programs entirely in machine language, I usually prefer to write in both Applesoft and machine language. This is because unless speed is required, Applesoft does offer some advantages in terms of program clarity and ease of modification. After all, if there were no advantage to Applesoft why would somebody have written it in the first place?

So, to that end, here are two new listings, the first in Applesoft:

```
10 IN$ = "X"
20 PRINT "ENTER THE STRING:";
30 CALL 768: IN$ = MID$(IN$,1)
40 IF IN$ = "END" THEN END
50 PRINT IN$: PRINT: GOTO 20
```

and the second in assembly language:

```
1 *****
2 * INPUT ROUTINE FOR FP BASIC *
3 *****
4 *
5 * IN$ = "" MUST BE 1ST VARIABLE
6 * DEFINED IN PROGRAM!
7 *
8   OBJ $300
9   ORG $300
10 *
```

```

11 GETLN EQU $FD6F
12 VARTAB EQU $69
13 BUFF EQU $200
14 *
15 *
0300:   A2 00   16 ENTRY LDX #$00
0302:   20 6F FD 17 JSR GETLN
0305:   A0 02   18 LDY #$02
0307:   8A     19 TXA
0308:   91 69   20 STA (VARTAB),Y
21 * STORE 'X-REG = LEN OF IN$'
22 * IN LEN BYTE OF IN$
23 *
030A:   C8     24 INY
030B:   A9 00   25 LDA #$00
030D:   91 69   26 STA (VARTAB),Y
030F:   C8     27 INY
0310:   A9 02   28 LDA #$02
0312:   91 69   29 STA (VARTAB),Y
30 * SET LOCATION PTR OF IN$ TO
31 * $200 (INPUT BUFFER)
32 *
0314:   8A 60   33 XFER TXA
0315:   A8     34 TAY ; Y-REG = LEN NOW
0316:   B9 00 02 35 X2 LDA BUFF,Y
0319:   29 7F   36 AND #$7F
031B:   99 00 02 37 STA BUFF,Y
031E:   88     38 DEY
031F:   C0 FF   39 CPY #$FF
0321:   D0 F3   40 BNE C2
41 *
0323:   60     42 DONE RTS

```

The important difference to notice here is that IN\$ has been defined as the first variable in the Applesoft program, and that the machine language routine uses this fact to transfer the string to Applesoft.

The way this is done begins at XFER. When an Applesoft string variable is stored, the name, length and location of the string are put in a table, whose beginning is pointed to by locations 69, 6A (VARTAB, VARTAB + 1).

Since IN\$ was the first variable defined, we know that its name and pointer will start at wherever VARTAB points. The name is held in positions 00 and 01, the length in 02, and the location in 03 and 04.

	+4	loc + 1
	+3	loc
	+2	length
LOMEM: + 1		INS
		I

←LOMEM: (\$69,6A)

By loading the Y-Register with #502, we can store the length of the entered string in the proper place. The location of INS is then set to \$200 by putting the appropriate bytes into positions 03 and 04. Now Applesoft is temporarily fooled into thinking that INS is at \$200—right where our input string is held!

The routine finishes by clearing the high bit, as before, and then returning with the RTS.

When the return is done, line 30 of the Applesoft program immediately assigns INS to itself in such a way as to force Applesoft to move INS from where it was in the input buffer to a new location up in its usual variable storage area. The net result can be obtained in various other ways besides the MIDS statement, but the way shown is the least intrusive in terms of affecting other variables. (You could use AS = INS:INS = AS, but then you'd need a second variable in your program—no problem, just more names to keep track of.)

Try assembling this routine and the Applesoft program. Make sure the input routine is loaded at \$300 before running the Applesoft program. Note that you can enter commas, quotes, control-C's, etc. Only END or pressing reset should be able to exit this routine.



CHAPTER 13

Reading/Writing Files on Disk

This chapter will challenge your devotion to the cause of learning assembly language programming.

Up until now the source listings have been very short and easily typed in a few minutes time. Unfortunately, the listings for this chapter are a bit longer than usual. But chin up! The result will be worth it! I've received quite a number of requests for information on how to read and write files on the disk. The programs listed will combine many of the techniques and routines you've learned so far into a single mini-database program.

The first program saves and loads the data by means of a simple BSAVE/BLOAD operation. This is fast and very straight forward. Here's the listing:

```
1 *****
2 * NAME FILE DEMO PROGRAM *
3 *****
4 *
5 *
6 OBJ $6000
7 ORG $6000
8 *
```

		9	HOME EQU \$FC58	
		10	COUT EQU \$FDED	
		11	RDKEY EQU \$FD0C	
		12	GETLN EQU \$FD6F	
		13	BUFF EQU \$200	
		14	VTAB EQU \$FC22	
		15	CH EQU \$24	
		16	CV EQU \$25	
		17	CTR EQU \$08	
		18	PTR EQU \$06	
		19	*	
		20	*	
6000:	A9 00	21	ENTRY LDA #\$500	
6002:	85 06	22	STA PTR	
6004:	A9 10	23	LDA #\$10	
6006:	85 07	24	STA PTR + 1	
6008:	A9 B1	25	LDA #\$B1	
600A:	85 08	26	STA CTR	
		27	*	
600C:	A0 00	28	CLR LDY #\$500	
600E:	91 06	29	STA (PTR),Y	
6010:	C8	30	INY	
6011:	A9 A0	31	LDA #\$A0	
6013:	91 06	32	STA (PTR),Y	
6015:	A9 00	33	LDA #\$500	
6017:	C8	34	INY	
6018:	91 06	35	STA (PTR),Y	
601A:	E6 07	36	INC PTR + 1	
601C:	E6 08	37	INC CTR	
601E:	A5 08	38	LDA CTR	
6020:	C9 B6	39	CMP #\$B6	
6022:	90 E8	40	BCC CLR	
		41	*	
		42	* PUTS '#1-5,SPC,00' IN BUFFER	
		43	*	
6024:	20 58 FC	44	MENU JSR HOME	
6027:	A9 02	45	P1 LDA #\$02	
6029:	85 25	46	STA CV ; VTAB 3	
602B:	20 22 FC	47	JSR VTAB	
602E:	20 C0 61	48	JSR PRINT	
6031:	B1 A9 A0	49	ASC "1) INPUT NAMES"	
603F:	8D 00	50	HEX 8D00	
		51	*	
6041:	A9 04	52	P2 LDA #\$04	
6043:	85 25	53	STA CV	
6045:	20 22 FC	54	JSR VTAB ; VTAB 5	
6048:	20 C0 61	55	JSR PRINT	
604B:	B2 A9 A0	56	ASC "2) PRINT NAMES"	
6059:	8D 00	57	HEX 8D00	
		58	*	

605B:	A9 06	59	P3 LDA # \$06
605D:	85 25	60	STA CV
605F:	20 22 FC	61	JSR VTAB ; VTAB 5
6062:	20 C0 61	62	JSR PRINT
6065:	B3 A9 A0	63	ASC "(3) SAVE NAMES"
6072:	8D 00	64	HEX 8D00
		65	*
6074:	A9 08	66	P4 LDA # \$08
6076:	85 25	67	STA CV
6078:	20 22 FC	68	JSR VTAB ; VTAB 9
607B:	20 C0 61	69	JSR PRINT
607E:	B4 A9 A0	70	ASC "(4) LOAD NAMES"
608B:	8D 00	71	HEX 8D00
		72	*
608D:	A9 0A	73	P5 LDA # \$0A
608F:	85 25	74	STA CV
6091:	20 22 FC	75	JSR VTAB ; VTAB 11
6094:	20 C0 61	76	JSR PRINT
6097:	B5 A9 A0	77	ASC "(5) END PROGRAM"
60A5:	8D 00	78	HEX 8D00
		79	*
60A7:	A9 0C	80	P6 LDA # \$0C
60A9:	85 25	81	STA CV
60AB:	20 22 FC	82	JSR VTAB ; VTAB 13
60AE:	20 C0 61	83	JSR PRINT
60B1:	D7 C8 C9	84	ASC "WHICH DO YOU WANT? "
60C4:	00	85	HEX 00
		86	*
60C5:	20 0C FD	87	M1 JSR RDKEY
60C8:	C9 B1	88	CMP # \$B1 ; '1'
60CA:	D0 06	89	BNE M2
60CC:	20 FB 60	90	JSR INPUT
60CF:	4C 24 60	91	JMP MENU
60D2:	C9 B2	92	M2 CMP # \$B2 ; '2'
60D4:	D0 09	93	BNE M3
60D6:	20 40 61	94	JSR DSPLY
60D9:	20 0C FD	95	JSR RDKEY
60DC:	4C 24 60	96	JMP MENU
60DF:	C9 B3	97	M3 CMP # \$B3 ; '3'
60E1:	D0 06	98	BNE M4
60E3:	20 76 61	99	JSR SAVE
60E6:	4C 24 60	100	JMP MENU
60E9:	C9 B4	101	M4 CMP # \$B4 ; '4'
60EB:	D0 06	102	BNE M5
60ED:	20 9E 61	103	JSR LOAD
60F0:	4C 24 60	104	JMP MENU
60F3:	C9 B5	105	M5 CMP # \$B5 ; '5'
60F5:	D0 01	106	BNE M6
60F7:	60	107	RTS
60F8:	4C 24 60	108	M6 JMP MENU

			109 *
			110 *
60FB:	20 40 61		111 INPUT JSR DSPLY ; SHOW WHAT'S THERE
			112 *
60FE:	A9 00		113 IO LDA #\$00
6100:	85 06		114 STA PTR
6102:	A9 10		115 LDA #\$10
6104:	85 07		116 STA PTR+1 ; SET PTR = \$1000
			117 *
6106:	A9 00		118 LDA #\$00
6108:	85 08		119 STA CTR
610A:	18		120 ILOOP CLC
610B:	A5 08		121 LDA CTR
610D:	65 08		122 ADC CTR
610F:	85 25		123 STA CV
6111:	20 22 FC		124 JSR VTAB
6114:	A9 00		125 LDA #\$00
6116:	85 24		126 STA CH
6118:	A8		127 TAY
6119:	20 27 61		128 JSR IP
611C:	E6 07		129 INC PTR+1
611E:	E6 08		130 INC CTR
6120:	A9 04		131 LDA #\$04
6122:	C5 08		132 CMP CTR
6124:	B0 E4		133 BCS ILOOP ; GET 5 NAMES
			134 *
6126:	60		135 IFIN RTS
			136 *
6127:	A2 00		137 IP LDX #\$00
6129:	20 6F FD		138 JSR GETLN
612C:	8A		139 TXA
612D:	F0 10		140 BEQ IPFIN ; EXIT IF <CR> ONLY
612F:	A8		141 TAY
6130:	A9 00		142 LDA #\$00
6132:	99 00 02		143 STA BUFFY ; PUT '0' AT END
6135:	B9 00 02		144 IPLOOP LDA BUFFY
6138:	91 06		145 STA (PTR),Y ; MOVE DATA TO PTR
			146 * ; BLOCK.
613A:	88		147 DEY
613B:	C0 FF		148 CPY #\$FF
613D:	D0 F6		149 BNE IPLOOP
613F:	60		150 IPFIN RTS
			151 *
6140:	20 58 FC		152 DSPLY JSR HOME
6143:	A9 00		153 LDA #\$00
6145:	85 08		154 STA CTR
			155 *
6147:	85 06		156 STA PTR
6149:	A9 10		157 LDA #\$10
614B:	85 07		158 STA PTR+1

614D:	18	159	D0 CLC
614E:	A5 08	160	LDA CTR
6150:	65 08	161	ADC CTR
6152:	85 25	162	STA CV ; VTAB (2*CTR) +1
6154:	20 22 FC	163	JSR VTAB
6157:	A9 00	164	LDA #\$00
6159:	85 24	165	STA CH ; HTAB 1
615B:	A8	166	TAY
		167	*
615C:	B1 06	168	DI LDA (PTR),Y
615E:	F0 06	169	BEQ D1FIN
6160:	20 ED FD	170	JSR COUT
6163:	C8	171	INY
6164:	D0 F6	172	BNE D1 (ALWAYS)
		173	*
6166:	A9 8D	174	D1FIN LDA #\$8D
6168:	20 ED FD	175	JSR COUT ; END W/ <CR>
616B:	E6 07	176	INC PTR + 1
616D:	E6 08	177	INC CTR
616F:	A9 04	178	LDA #\$04
6171:	C5 08	179	CMP CTR
6173:	B0 D8	180	BCS D0 ; PRINT 5 NAMES
		181	*
6175:	60	182	DSFIN RTS
		183	*
		184	*
6176:	A9 8D	185	SAVE LDA #\$8D
6178:	20 ED FD	186	JSR COUT ; CLEAR OUTPUT BUFFER
617B:	20 C0 61	187	OPEN JSR PRINT
617E:	84	188	HEX 84
617F:	C2 D3 C1	189	ASC "BSAVE DEMO FILE,A\$1000,L\$500"
619B:	8D 00	190	HEX 8D00
		191	*
619D:	60	192	SFIN RTS
		193	*
		194	*
619E:	A9 8D	195	LOAD LDA #\$8D
61A0:	20 ED FD	196	JSR COUT
		197	*
61A3:	20 C0 61	198	JSR PRINT
61A6:	84	199	HEX 84
61A7:	C2 CC CF	200	ASC "BLOAD DEMO FILE,A\$1000"
61BD:	8D 00	201	HEX 8D00
		202	*
61BF:	60	203	RTS
		204	*
		205	*
		206	*
61C0:	68	207	PRINT PLA
61C1:	85 06	208	STA PTR

61C3:	68	209	PLA
61C4:	85 07	210	STA PTR + 1
61C6:	A0 01	211	LDY # \$0
61C8:	B1 06	212	P0 LDA (PTR), Y
61CA:	F0 06	213	BEQ PFIN
61CC:	20 ED FD	214	JSR COUT
61CF:	C8	215	INY
61D0:	D0 F6	216	BNE P0 (ALWAYS)
		217	*
61D2:	18	218	PFIN CLC
61D3:	98	219	TYA
61D4:	65 06	220	ADC PTR
61D6:	85 06	221	STA PTR
61D8:	A5 07	222	LDA PTR + 1
61DA:	69 00	223	ADC # \$00
61DC:	48	224	PHA
61DD:	A5 06	225	LDA PTR
61DF:	48	226	PHA
61E0:	60	227	EXIT RTS
		228	*
		229	*
61E1:	00	230	EOF BRK
		231	*
		232	*

To understand how it works, consider these conditions:

Data will be stored in the area from \$1000–\$14FF. This area is called a *buffer*. A total of five strings will be stored, each beginning at an exact page boundary (\$1000, 1100, 1200, etc.). It is assumed that no string will be longer than 255 bytes—a fairly safe assumption since the input routine won't allow this either.

A zero page pointer (cleverly labeled PTR) will be used to control which range in the buffer is currently being accessed for a particular string.

The basic routines used to make the overall idea work are as follows:

- 1) An input routine using the Monitor (\$FD6F = GETLN).
- 2) A print routine using a JSR and a stack manipulation. (Not the DATA type.)
- 3) A single key input routine present in the Monitor used to get the command key (\$FD0C = RDKEY).
- 4) The execution of DOS commands from machine language by preceding phrases with a control-D.

To use the program, either BRUN the final code or call it

directly from Basic with a CALL 24576, or from the Monitor with a 6000G.

A menu will appear with these choices:

- 1) INPUT NAMES
- 2) PRINT NAMES
- 3) SAVE NAMES
- 4) LOAD NAMES
- 5) END PROGRAM

To try the routine out, use 1 to enter five sample names. Then use 2 to view the data you've entered. You may then use 3 to save the data as a binary file on a diskette. Then rerun the program, and verify that only the numbers 1 through 5 exist in the buffer (option 2). Then retrieve your data by using the LOAD command (option 4), and view again to confirm a successful load.

In detail, this is how the program works:

At entry, PTR is set to point to \$1000, where the name buffer begins. The Accumulator is then loaded with with ASCII value for the character 1, and the CLR routine entered.

CLR puts the characters 1 through 5 into each of the string spaces. Each digit is followed by a space, and then a 0. I used 0 as an end-of-string marker, but the choice is somewhat arbitrary.

MENU clears the screen and presents the user with the available choices. Points of interest here are the VTAB operation and the print routine. To VTAB to a given line from machine language, one of the easiest ways is to load CV with the line you wish to go to, and then JSR to the Monitor's VTAB routine (\$FC22). Normally, we might also wish to either print a carriage return, or set CH to zero. Note that CV and CH are the computer's vertical and horizontal cursor position bytes, as used by the Monitor. You can always tell the cursor position by examining these bytes, and CH may be forced to a desired value to accomplish the same as an HTAB in Basic.

The print routine is the one described in chapter twelve, and is useful because the JSR PRINT can be immediately followed with the data to print. This is more similar to the Basic PRINT statement, and also avoids setting up a lot of specific data tables to do the printing.

Once the menu is printed on the screen, line 87 of the source file does the JSR to RDKEY. This gets the command key from the user, which is then tested by the M1 to M6 series of checks.

After calling `RKEY`, the keyboard value was returned in the Accumulator, and we can directly test to see which key was pressed. The key is then compared with one of the five desired responses. If no match is found, it jumps back to `MENU` to repeat the display and command input. Other than reset, 5 is the only way to exit the program.

Let's examine the menu choices:

If you enter 1, control is directed to the section labeled `INPUT`. The first thing done there is to `JSR DSPLY`. At this point, it's only necessary to understand that `DSPLY` just clears the screen and shows the five strings currently in memory.

After `DSPLY`, `PTR` is initialized to point to the beginning of the buffer (`$1000`), and the counter set to zero. The main input loop comes next. Here `CTR` is used to calculate what line (vertical position) to put the cursor on. (`DSPLY` used the same algorithm to display present data.) After `VTAB`, the equivalent of `HTAB 0` is done, followed by the jump to the actual input routine, here labeled `IP`. This is the routine from the previous chapter that gets a line and then moves it to a location indicated by `PTR`.

There are a few subtle items in the `IP` routine that should be noted. The first is line 140. If return alone is entered (i.e. no new data), the routine immediately returns without rewriting the old string. This is to allow editing of a single entry by skipping the entries not of interest. Try it to see how it works.

The second item is the characteristic of this particular input routine to put the trailing zero at the end of the line. This is done on lines 141-143.

When it returns from `IP`, the counter is incremented and checked to see if it exceeds `#$04`. If not, `LOOP` repeats until five strings are input. After the fifth string is entered, the program returns to the menu.

If choice 2 is entered, the `DSPLY` routine is called. The sole purpose of this section is to clear the screen and print the five names in memory. At entry to `DSPLY`, a `JSR $FC58` does a `HOME` and the `CTR` is initialized to zero. As in the `INPUT` section, `CTR` is then used to calculate the `VTAB` position to print each line.

`DI` is the part that actually prints each line by scanning (and outputting through `COUT`) all the bytes at each range indicated by `PTR`. Note that as a safety check, if a zero did not happen to be present due to some other error, eventually the Y-Register will pass `#$FF` and the program will fall through to `DIFIN`.

`DIFIN` provides an ending carriage return to the string, and

then increments CTR until all five strings have been printed.

The load/save operations are quite simple. Knowing where the buffer is located, the entire block is accessed by doing either a BLOAD or BSAVE. Remember that disk commands are done from machine language just as they would be done from Basic. The program need only output a control-D followed by a legal DOS command and a carriage return. Again the print routine is used to facilitate this.

If choice 5 is entered, then an RTS is executed to end the program.

Reading and Writing Text Files

This second listing is basically a modification of the first program. If you wish, rather than retype the entire file, you can just edit the first listing to add lines 20-29, and 194-228.

```
1 *****
2 * NAME FILE DEMO PROGRAM 2 *
3 *****
4 *
5 *
6   OBJ $6000
7   ORG $6000
8 *
9 HOME EQU $FC58
10 COUT EQU $FDED
11 RDKEY EQU $FD0C
12 GETLN EQU $FD6F
13 BUFF EQU $200
14 VTAB EQU $FC22
15 CH EQU $24
16 CV EQU $25
17 CTR EQU $08
18 PTR EQU $06
19 *
20 PROMPT EQU $33
21 CURLIN EQU $75
22 LANG EQU $AAB6
23 REENTRY EQU $3D0
24 *
6000:  A9 40      25 ENTRY LDA #$40
6002:  8D B6 AA  26   STA LANG ; LANG = FP
6005:  85 76      27   STA CURLIN+1 ; RUNNING PROG
6007:  A9 06      28   LDA #$06
6009:  85 33      29   STA PROMPT ; NOT DIRECT MODE
600B:  A9 00      30   LDA #$00
```

600D:	85 06	31	STA PTR
600F:	A9 10	32	LDA #\$10
6011:	85 07	33	STA PTR + 1
6013:	A9 B1	34	LDA #\$B1
6015:	85 08	35	STA CTR
		36	*
6017:	A0 00	37	CLR LDY#\$00
6019:	91 06	38	STA (PTR),Y
601B:	C8	39	INY
601C:	A9 A0	40	LDA #\$A0
601E:	91 06	41	STA (PTR),Y
6020:	A9 00	42	LDA #\$00
6022:	C8	43	INY
6023:	91 06	44	STA (PTR),Y
6025:	E6 07	45	INC PTR + 1
6027:	E6 08	46	INC CTR
6029:	A5 08	47	LDA CTR
602B:	C9 B6	48	CMP #\$B6
602D:	90 E8	49	BCC CLR
		50	*
		51	* PUTS '#1-5,SPC,00' IN BUFFER
		52	*
602F:	20 58 FC	53	MENU JSR HOME
6032:	A9 02	54	P1 LDA #\$02
6034:	85 25	55	STA CV ; VTAB 3
6036:	20 22 FC	56	JSR VTAB
6039:	20 0A 62	57	JSR PRINT
603C:	B1 A9 A0	58	ASC "(1) INPUT NAMES"
604A:	8D 00	59	HEX 8D00
		60	*
604C:	A9 04	61	P2 LDA #\$04
604E:	85 25	62	STA CV
6050:	20 22 FC	63	JSR VTAB ; VTAB 5
6053:	20 0A 62	64	JSR PRINT
6056:	B2 A9 A0	65	ASC "(2) PRINT NAMES"
6064:	8D 00	66	HEX 8D00
		67	*
6066:	A9 06	68	P3 LDA #\$06
6068:	85 25	69	STA CV
606A:	20 22 FC	70	JSR VTAB ; VTAB 7
606D:	20 0A 62	71	JSR PRINT
6070:	B3 A9 A0	72	ASC "(3) SAVE NAMES"
607D:	8D 00	73	HEX 8D00
		74	*
607F:	A9 08	75	P4 LDA #\$08
6081:	85 25	76	STA CV
6083:	20 22 FC	77	JSR VTAB ; VTAB 9
6086:	20 0A 62	78	JSR PRINT
6089:	B4 A9 A0	79	ASC "(4) LOAD NAMES"
6096:	8D 00	80	HEX 8D00

		81	*
6098:	A9 0A	82	P5 LDA #\$0A
609A:	85 25	83	STA CV
609C:	20 22 FC	84	JSR VTAB ; VTAB 11
609F:	20 0A 62	85	JSR PRINT
60A2:	B5 A9 A0	86	ASC "5) END PROGRAM"
60B0:	8D 00	87	HEX 8D00
		88	*
60B2:	A9 0C	89	P6 LDA #\$0C
60B4:	85 25	90	STA CV
60B6:	20 22 FC	91	JSR VTAB ; VTAB 13
60B9:	20 0A 62	92	JSR PRINT
60BC:	D7 C8 C9	93	ASC "WHICH DO YOU WANT? "
60CF:	00	94	HEX 00
		95	*
60D0:	20 0C FD	96	M1 JSR RDKEY
60D3:	C9 B1	97	CMP #\$B1 ; '1'
60D5:	D0 06	98	BNE M2
60D7:	20 08 61	99	JSR INPUT
60DA:	4C 2F 60	100	JMP MENU
60DD:	C9 B2	101	M2 CMP #\$B2 ; '2'
60DF:	D0 09	102	BNE M3
60E1:	20 4D 61	103	JSR DSPLY
60E4:	20 0C FD	104	JSR RDKEY
60E7:	4C 2F 60	105	JMP MENU
60EA:	C9 B3	106	M3 CMP #\$B3 ; '3'
60EC:	D0 06	107	BNE M4
60EE:	20 83 61	108	JSR SAVE
60F1:	4C 2F 60	109	JMP MENU
60F4:	C9 B4	110	M4 CMP #\$B4 ; '4'
60F6:	D0 06	111	BNE M5
60F8:	20 C7 61	112	JSR LOAD
60FB:	4C 2F 60	113	JMP MENU
60FE:	C9 B5	114	M5 CMP #\$B5 ; '5'
6100:	D0 03	115	BNE M6
6102:	4C D0 03	116	JMP REENTRY
6105:	4C 2F 60	117	M6 JMP MENU
		118	*
		119	*
6108:	20 4D 61	120	INPUT JSR DSPLY ; SHOW WHAT'S THERE
		121	*
610B:	A9 00	122	I0 LDA #\$00
610D:	85 06	123	STA PTR
610F:	A9 10	124	LDA #\$10
6111:	85 07	125	STA PTR+1 ; SET PTR = \$1000
		126	*
6113:	A9 00	127	LDA #\$00
6115:	85 08	128	STA CTR
6117:	18	129	ILOOP CLC
6118:	A5 08	130	LDA CTR

611A:	65	08	131	ADC CTR
611C:	85	25	132	STA CV
611E:	20	22 FC	133	JSR VTAB
6121:	A9	00	134	LDA # \$00
6123:	85	24	135	STA CH
6125:	A8		136	TAY
6126:	20	34 61	137	JSR IP
6129:	E6	07	138	INC PTR + 1
612B:	E6	08	139	INC CTR
612D:	A9	04	140	LDA # \$04
612F:	C5	08	141	CMP CTR
6131:	B0	E4	142	BCS ILOOP ; GET 5 NAMES
			143	*
6133:	60		144	IFIN RTS
			145	*
6134:	A2	00	146	IP LDX # \$00
6136:	20	6F FD	147	JSR GETLN
6139:	8A		148	TXA
613A:	F0	10	149	BEQ IPFIN ; EXIT IF <CR> ONLY
613C:	A8		150	TAY
613D:	A9	00	151	LDA # \$00
613F:	99	00 02	152	STA BUFF,Y
6142:	B9	00 02	153	IPLOOP LDA BUFF,Y ; PUT '0' AT END
6145:	91	06	154	STA (PTR),Y ; MOVE DATA TO PTR
			155	* ; BLOCK.
6147:	88		156	DEY
6148:	C0	FF	157	CPY # \$FF
614A:	D0	F6	158	BNE IPLOOP
614C:	60		159	IPFIN RTS
			160	*
614D:	20	58 FC	161	DSPLY JSR HOME
6150:	A9	00	162	LDA # \$00
6152:	85	08	163	STA CTR
			164	*
6154:	85	06	165	STA PTR
6156:	A9	10	166	LDA # \$10
6158:	85	07	167	STA PTR + 1
615A:	18		168	D0 CLC
615B:	A5	08	169	LDA CTR
615D:	65	08	170	ADC CTR
615F:	85	25	171	STA CV ; VTAB (2*CTR) + 1
6161:	20	22 FC	172	JSR VTAB
6164:	A9	00	173	LDA # \$00
6166:	85	24	174	STA CH ; HTAB 1
6168:	A8		175	TAY
			176	*
6169:	B1	06	177	D1 LDA (PTR),Y
616B:	F0	06	178	BEQ D1FIN
616D:	20	ED FD	179	JSR COUT
6170:	C8		180	INY

6171:	D0 F6	181	BNE D1 (ALWAYS)
		182	*
6173:	A9 8D	183	D1FIN LDA #\$8D
6175:	20 ED FD	184	JSR COUT ; END W/ <C/R>
6178:	E6 07	185	INC PTR+1
617A:	E6 08	186	INC CTR
617C:	A9 04	187	LDA #\$04
617E:	C5 08	188	CMP CTR
6180:	B0 D8	189	BCS D0 ; PRINT 5 NAMES
		190	*
6182:	60	191	DSFIN RTS
		192	*
		193	*
6183:	A9 8D	194	SAVE LDA #\$8D
6185:	20 ED FD	195	JSR COUT ; CLEAR OUTPUT
6188:	20 0A 62	196	OPENW JSR PRINT
618B:	84	197	HEX 84 ; CTRL-D
618C:	CF D0 C5	198	ASC "OPEN NAME TEXT FILE"
619F:	8D 84	199	HEX 8D84
61A1:	D7 D2 C9	200	WRITE ASC "WRITE NAME TEXT FILE"
61B5:	8D 00	201	HEX 8D00
		202	*
61B7:	20 4D 61	203	SVLOOP JSR DSPLY ; PRINT NAMES TO DISK
		204	*
61BA:	20 0A 62	205	CLOSEW JSR PRINT
61BD:	8D 84	206	HEX 8D84
61BF:	C3 CC CF	207	ASC "CLOSE"
61C4:	8D 00	208	HEX 8D00
61C6:	60	209	SVFIN RTS
		210	*
		211	*
61C7:	A9 8D	212	LOAD LDA #\$8D
61C9:	20 ED FD	213	JSR COUT
		214	*
61CC:	20 0A 62	215	OPENR JSR PRINT
61CF:	84	216	HEX 84
61D0:	CF D0 C5	217	ASC "OPEN NAME TEXT FILE"
61E3:	8D 84	218	HEX 8D84
61E5:	D2 C5 C1	219	READ ASC "READ NAME TEXT FILE"
61F8:	8D 00	220	HEX 8D00
		221	*
61FA:	20 0B 61	222	RDLOOP JSR IO ; GET NAMES FROM DISK
		223	*
61FD:	20 0A 62	224	CLOSER JSR PRINT
6200:	8D 84	225	HEX 8D84
6202:	C3 CC CF	226	ASC "CLOSE"
6207:	8D 00	227	HEX 8D00
6209:	60	228	RDFIN RTS
		229	*
		230	*

		231 *
620A:	68	232 PRINT PLA
620B:	85 06	233 STA PTR
620D:	68	234 PLA
620E:	85 07	235 STA PTR + 1
6210:	A0 01	236 LDY #\$01
6212:	B1 06	237 P0 LDA (PTR),Y
6214:	F0 06	238 BEQ PFIN
6216:	20 ED FD	239 JSR COUT
6219:	C8	240 INY
621A:	D0 F6	241 BNE P0 (ALWAYS)
		242 *
621C:	18	243 PFIN CLC
621D:	98	244 TYA
621E:	65 06	245 ADC PTR
6220:	85 06	246 STA PTR
6222:	A5 07	247 LDA PTR + 1
6224:	69 00	248 ADC #\$00
6226:	48	249 PHA
6227:	A5 06	250 LDA PTR
6229:	48	251 PHA
622A:	60	252 EXIT RTS
		253 *
		254 *
622B:	00	255 EOF BRK
		256 *
		257 *

The theory to the second program is fairly simple. If you think about it, the INPUT and DPLY sections are essentially equivalent to a FOR I = 1 TO 5 / NEXT I type loop that respectively inputs and prints five strings. In a Basic program, all that would be required to access a text file would be to precede the execution of those routines with the OPEN, READ and the OPEN, WRITE commands. (I'm assuming you're familiar with the normal access of Apple DOS text files. If not, read your manual!)

If you examine the new save and load routines you'll notice two changes. First, rather than printing BSAVE or BLOAD, the files are OPENED and the READ or WRITE command output. Notice that each command begins with a control-D and ends with a carriage return. Second, after the command is printed, a JSR is done to the IP or DPLY routine as is appropriate. Last of all, a CLOSE is output before returning to the menu.

According to what we know so far, these should be the only changes necessary to access text files. There is one last catch though.

Apple DOS complicates things by not allowing the user to

OPEN text files from the immediate mode. When a machine language program is running, DOS thinks we're still in the immediate mode and won't let us access the text files. What's needed is a way to fool DOS into thinking we're running a program.

This is done by using three internal management locations in the Apple. LANG (\$AAB6) is what DOS uses to keep track of which language is currently running. CURLIN (\$75,76) is Applesoft's register for the bytes of the line number of the program currently being executed. In the immediate mode, the high-order byte (\$76) defaults to #\$FF. Applesoft can tell if a program is running by looking for a non-#\$FF value in this location. The other way it knows a program is running is to check the location (\$33) that holds the ASCII value for the prompt character. In the immediate mode of Applesoft, this is #\$DD, equivalent to the 'J' character. In a running program, this changes to #\$06.

To fool DOS, all we need do is load these three locations at the beginning of the routine. Finally, when exiting the program, rather than a simple RTS, a JMP \$3D0 is done to do a soft reentry to Basic. This will store the bytes we've altered to fool DOS and also return us to the current language.¹

Try these programs out. You'll find they make an excellent summary of many of the ideas and routines discussed so far, and also will provide a valuable model for your own programs.

1. Some people have also inquired as to whether the check for a write-protect label can be defeated by modifying DOS. The answer is yes and no. Yes, the part of the code that generates the error can be eliminated, but because the write-protect switch is physically wired into the recording head write system, you cannot defeat it without actually removing or altering the switch itself.

///

///

///

///

///

///

///

///

///

///

///

///

///

///

///



CHAPTER 14

Special Programming Techniques

It has long been my feeling that it is not enough to just know an arbitrary selection of options or commands when using any tool, program, or programming language. Equally important are the techniques with which the options are combined to achieve the desired results.

With time and practice you will develop your own skills at creating efficient machine language routines, but that process can be assisted by examining the techniques that others have developed in previous programming efforts.

I have tried in this book to provide a reasonable mix of programming techniques, along with the usual ration of new commands.

Relocatable versus Nonrelocatable Code

In chapter twelve, I presented two print subroutines for the output of text to the screen or disk text file. The disadvantage of both routines was that they were not *relocatable*. To see what this means, consider the following program:

```
1 *****  
2 * NONRELOC PRINT DEMO *  
3 *****
```

		4	*
		5	*
		6	OBJ \$300
		7	ORG \$300
		8	COUT EQU \$FDED
		9	*
		10	*
0300:	20 0D 03	11	ENTRY JSR PRNT
		12	*
0303:	4C 0C 03	13	DONE JMP EXIT
		14	*
0306:	D4 C5 D3	15	DATA ASC "TEST"
030A:	8D 00	16	HEX 8D00
		17	*
030C:	60	18	EXIT RTS
		19	*
030D:	A2 00	20	PRNT LDX #\$00
030F:	BD 06 03	21	LOOP LDA DATA,X
0312:	F0 EF	22	BEQ DONE
0314:	20 ED FD	23	JSR COUT
0317:	E8	24	INX
0318:	D0 F5	25	BNE LOOP
031A:	60	26	FIN RTS

This program, as written, can only be run at the location specified by the `ORG` statement, in this case \$300. Thus it is called *nonrelocatable* code. Machine code becomes nonrelocatable through the use of any statements which involve absolute addressing. The most common examples are the `JMP` and `JSR` commands, and the use of data statements, usually in print routines.

The first statement of this type occurs on line 11. The `JSR` to `PRNT` (\$30D) will only work so long as `PRNT` is at \$30D. If the routine were to be loaded into memory at \$400 (instead of \$300), the routine would take the `JSR` to a block of nonexistent code at \$30D.

Likewise, the `JMP` on line 13 has the same difficulty as does the `DATA,X` statement on line 21. Any attempt to run the code at an address other than \$300 will result in disaster.

It should be noted however that not all `JSRs` and `JMPs` are universally troublesome. The `JSR COUT` (\$FDED) will execute properly no matter where the object code is located since the reference is to a location outside of the object code block.

The general rule then is that any code which makes reference to absolute addresses within itself will not be relocatable, whereas code that does not suffer from this limitation can be run anywhere in memory.

The problem of relocatability may seem slight since any given routine is usually designed to be put at a definite location (usually either at \$300 or at the top of memory) and then protected via the Applesoft HIMEM: statement. However, as the number of routines you use increases, you will encounter more and more conflicts between routines originally written to occupy the same memory ranges. In addition, it is also occasionally desirable to directly append machine code to the end of Applesoft programs, where they will float up and down in memory at the end of the Basic portion of the listing, being automatically moved as lines are added or deleted.

For these reasons, it is in the long run better to write code to run anywhere in memory when possible, thus avoiding future headaches about where to put everything.

The remainder of this chapter will discuss the various ways of avoiding the use of absolute addressing, thus creating code that can be used anywhere in memory, regardless of the ORG statement used at assembly time.

JMP Commands

This is an example of a common use of the JMP command to jump over a range of memory, here represented by the FILL section. At the destination, EXPT, the bell routine is called as a trivial example of where a subroutine might be executed.

```

1 *****
2 * NONRELOC JMP DEMO *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8   BELL EQU $FF3A
9 *
10 *
0300:  4C 04 03 11 ENTRY JMP EXPT
12 *
0303:  EA      13 FILL NOP
14 *
0304:  20 3A FF 15 EXPT JSR BELL
16 *
0307:  60      17 DONE RTS

```

An alternative to this is the use of a forced branch statement, as shown in this example:

```

1 *****
2 * RELOCATABLE JMP 1 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8   BELL EQU $FF3A
9 *
10 *
0300: 18   11 ENTRY CLC
0301: 90 01 12   BCC EXPT
13 *
0303: EA   14 FILL NOP
15 *
0304: 20 3A FF 16 EXPT JSR BELL
17 *
0307: 60   18 DONE RTS
```

Notice that by clearing the carry, and then immediately executing the BCC, the same result is obtained as when the JMP command was used in the earlier listing.

The main caution to observe is that the jump is not made over a distance of greater than 127 bytes, although most assemblers will give an error at assembly time if this is attempted. In addition, since the carry is cleared to force the branch, routines that set or clear the carry to indicate certain conditions may have compatibility problems with this approach.

Both limitations can be solved by slight modifications to this listing. The first is by using the overflow flag, often represented by a V. You should remember that the Status Register of the 6502 contains certain flags that are conditioned by various operations. These flags can be checked and appropriate responses made, depending on their status. Examples of flags already covered are the carry and zero flags.

The overflow flag is another bit in the Status Register which is set either by the BIT command (The overflow flag is set to bit 6 of the memory location), or by an ADC command. The overflow will be set whenever there is a carry from bit 6 to bit 7 as a result of an ADC operation.

These details are mentioned only in passing at this point, and you need not be concerned if it is not entirely clear. The main

reason for bringing it up is that the overflow flag is used much more infrequently than the carry, and thus is a slightly more desirable flag to use when creating a forced branch.

To make jumps over distances greater than 127 bytes, a *stepping* technique can be used. This is done by creating a series of the branch commands throughout the code to facilitate the program flow from one part to another. It is generally not too difficult to find breaks between routines to insert the branch statements required for the stepping action.

Both techniques are illustrated here:

```

1 *****
2 * RELOCATABLE JMP 2 *
3 *****
4 *
5 *
6   OBJ $300
7   ORG $300
8   BELL EQU $FF3A
9 *
10 *
0300:  B8      11  ENTRY CLV
0301:  50 01   12   BVC STEP
13 *
0303:  EA      14  FILL1 NOP
15 *
0304:  50 01   16  STEP BVC EXPT
17 *
0306:  EA      18  FILL2 NOP
19 *
0307:  20 3A FF 20  EXPT JSR BELL
21 *
030A:  60      22  DONE RTS
```

Although only one step is shown here, any number may be used, depending on what is required to span the required distance.

Determining Code Location

Solving the JMP problem is only the beginning of the task. Very often it is important to know just where in memory the code is currently being run. One example of this is the code present on the disk controller cards. Since the card can be put in one of seven slots, and since each slot occupies a unique memory

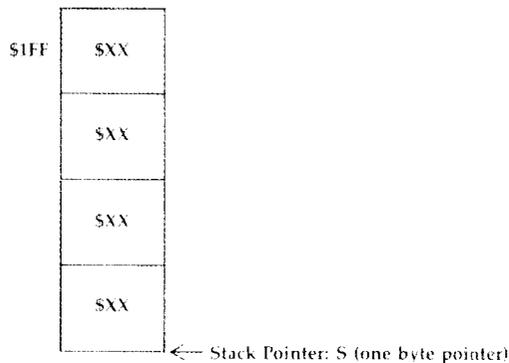
range, some technique is required to answer the question, "Where are we?"

		1 *****
		2 * LOCATOR 1 *
		3 *****
		4 *
		5 OBJ \$300
		6 ORG \$300
		7 *
		8 PTR EQU \$06
		9 RTRN EQU \$FF58
		10 STCK EQU \$100
		11 *
0300:	20 58 FF	12 ENTRY JSR RTRN
0303:	BA	13 TSX
0304:	BD 00 01	14 LDA STCK,X
0307:	85 07	15 STA PTR+1
0309:	CA	16 DEX
030A:	BD 00 01	17 LDA STCK,X
030D:	85 06	18 STA PTR
030F:	60	19 DONE RTS

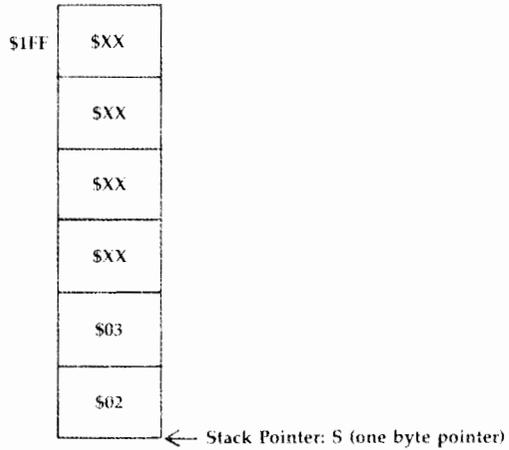
The success of this routine is based entirely on the predictable nature of the stack, and its function when a JSR is executed.

The stack was briefly described in chapter eight. At this point a little greater detail is necessary. The stack is a reserved part of memory from \$100 to \$1FF. It is used as a temporary holding buffer for various kinds of information required by the 6502 microprocessor. Information put on the stack is always retrieved in the opposite order from which it was deposited. This is often called LIFO. The analogy of a stack of plates was used earlier, but the time has come to examine what actually occurs.

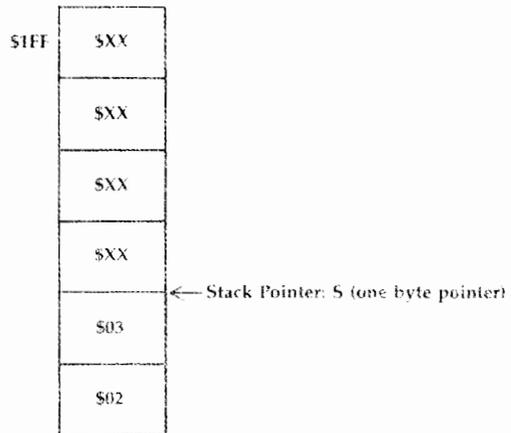
Before the JSR \$FF58:



During the JSR \$FF58:



After the JSR \$FF58:



Whenever a JSR is done, the stack is used to hold the address to which the return should be made when the expected RTS is encountered. The preceding diagrams illustrate this. Location \$FF58 is a simple RTS in the Monitor ROM which will be used to set up a *dummy* return address. Before the JSR, the stack pointer is set to some arbitrary position in the stack. Upon executing the JSR, the return address of \$302 is put on the stack and the stack pointer is decremented two bytes. Note that the stack stores the

data from the top down, advancing the pointer as new data is added. When the RTS is encountered (immediately in the case of \$FF58), the stack pointer is returned to its original position and the return made.

Note that the address stored, \$302, is the last byte of the JSR command, or put another way, one byte less than the address of the next immediate command following the JSR.

Upon return from the JSR, the stack pointer is transferred to the X-Register with the TSX command on line 13. Because the stack pointer is at the next available byte on the stack, this will also point at the high-order byte of the return address still left in memory there. This is retrieved with the LDA STCK,X on line 15 and put in a temporary pointer location PTR + 1 (\$07). The X-Register is then decremented and the low-order byte retrieved and put in PTR (\$06).

The final RTS of the routine returns control to the caller, at which point \$06,07 may be examined to verify the successful determination of the address \$302. You may wish to run this routine at several different locations in memory to verify that in each case PTR is properly set to (ENTRY + 2).

What you have then is a short routine which can determine where in memory it is currently being run. The only disadvantage to this routine is that the high-order byte is retrieved first, thus complicating things if we want to add some offset value to the return address. The desirability of this will be shown shortly. In the mean time, consider this altered version of the Locator 1 routine:

```

1 *****
2 * LOCATOR 2 *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PTR EQU $06
9   RTRN EQU $FF58
10  STCK EQU $100
11 *
0300:  20 58 FF 12  ENTRY JSR RTRN
0303:  BA          13   TSX
0304:  CA          14   DEX
0305:  BD 00 01 15   LDA STCK,X
0308:  85 06          16   STA PTR
```

```

030A:  E8          17  INX
030B:  BD 00 01   18  LDA STCK,X
030E:  85 07      19  STA PTR+1
0310:  60          20  DONE RTS

```

What I've done here is decrement the X-Register (line 14) immediately after the TSX statement so that the low-order byte of the address can be retrieved first. The INX is then later used to go back and get the high-order byte. The advantage of this system is that it makes adding an offset much easier.

To show what we can now do, look at this revised print routine:

```

1  *****
2  * RELOCATABLE PRINT 1 *
3  *****
4  *
5  OBJ $300
6  ORG $300
7  *
8  PTR EQU $06
9  COUT EQU $FDED
10 RTRN EQU $FF58
11 STCK EQU $100
12 *
13 *
14 ENTRY JSR RTRN
0300:  20 58 FF   15  CLV
0303:  B8          16  BVC CONT
0304:  50 06          17  *
0306:  D4 C5          18  DATA ASC "TEST"
030A:  BD 00          19  HEX 8D00
030C:  BA          20  *
030D:  CA          21  CONT TSX
030E:  18          22  DEX
030F:  BD 00 01     23  CLC
0312:  69 04          24  LDA STCK,X
0314:  85 06          25  ADC #$04
0316:  E8          26  STA PTR
0317:  BD 00 01     27  INX
031A:  69 00          28  LDA STCK,X
031C:  85 07          29  ADC #$00
031E:  A0 00          30  STA PTR+1
0320:  B1 06          31  *
0322:  F0 06          32  PRNT LDY #$00
0324:  20 ED FD      33  LOOP LDA (PTR),Y
0327:  C8          34  BEQ FIN
0328:  20 ED FD      35  JSR COUT
0329:  C8          36  INY

```

```

0328:   D0 F6          37   BNE LOOP ; (ALWAYS 'TILL 255)
                                38   *
032A:   60            39   FIN RTS

```

After calling the dummy return statement, a forced branch over the data section is done. This will have no effect on the address remaining on the stack. At CONT, we take the general procedure used in Locator 2, and add the CLC and ADC statements needed to add an offset to the address on the stack. What we need is the starting address of the ASCII data to be printed. Since the data starts at \$306 and the address on the stack is \$302 (see earlier examples) the offset needed is \$04.

This may seem arbitrary, but the value to add will always be \$04 if you always do the CLV, BVC \$XXXX branch immediately after the return, and follow that with the data to be printed.

Once the actual address of the ASCII data has been calculated, it is printed in the PRNT section by use of the indexed pointer at LOOP.

JSR Simulations

You might get the impression from the above example that a tremendous code expansion takes place to accomplish the relocatability of a program. This is somewhat true but depends on how you write the program. The use of CLV, BVC \$XXXX only takes three bytes where the JMP \$XXXX it was replacing also used three bytes.

The stack operations just discussed take a small number of bytes to implement, but could become rather large if used many times. What is needed is a way to put the stack operations in a subroutine. Unfortunately, the JSR is one of the nonrelocatable commands.

```

1 *****
2 * NONRELOC. JSR DEMO *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8 BELL EQU $FF3A
9 *
10 *
0300:  20 04 03   11 ENTRY JSR TEST
12 *

```

```

0303: 60          13 DONE RTS
          14 *
0304: EA          15 TEST NOP
          16 *
0305: 20 3A FF    17 EXPT JSR BELL
          18 *
0306: 60          19 FIN RTS
          20 *
          21 * WILL RETURN TO DONE
          22 *

```

This routine is very similar to the nonrelocatable `JMP` demo presented earlier, with the exception that the call to the `BELL` routine has been made a subroutine itself, headed by the label `TEST`. In this listing, `TEST` is followed by a dummy `NOP` statement, but we'll fill that in shortly.

This program, as written, can only be run at the address specified in the `ORG` statement. Here is an improved version, using a simulation of the `JSR` command:

```

          1 *****
          2 * RELOCATABLE JSR SIM *
          3 *****
          4 *
          5 OBJ $300
          6 ORG $300
          7 *
          8 PTR EQU $06
          9 BELL EQU $FF3A
         10 RTRN EQU $FF58
         11 STCK EQU $100
         12 *
         13 *
0300: 20 58 FF    14 ENTRY JSR RTRN
0303: B8          15 CLV
0304: 50 01      16 BVC TEST
          17 *
0306: 60          18 DONE RTS
          19 *
0307: BA          20 TEST TSX
0308: CA          21 DEX
0309: 18          22 CLC
030A: BD 00 01   23 LDA STCK,X
030D: 69 03      24 ADC #$03
030F: 85 06      25 STA PTR
0311: E8          26 INX
0312: BD 00 01   27 LDA STCK,X
0315: 69 00      28 ADC #$00

```

```

0317:  85 07          29 STA PTR + 1
                                30 *
0319:  20 3A FF        31 EXPT JSR BELL
                                32 *
031C:  A5 07          33 FIX LDA PTR + 1
031E:  48              34 PHA
031F:  A5 06          35 LDA PTR
0321:  48              36 PHA
0322:  60              37 FIN RTS
                                38 *
                                39 * WILL RETURN TO DONE.
                                40 *

```

This program is very similar to the Print 1 program, with two exceptions. First, #503 is added instead of #504 to the address on the stack. This is a subtle point worth mentioning, and you should review the listings until you feel comfortable with what is being done. Remember that the return address for a JSR/RTS is always *one less* than the address you want to return to. In the case of the DATA statement, we needed to know the exact address of the first character of the string to be printed. Hence the difference in the offset value used in each case.

Once the offset has been added and the proper return address calculated, the FIX section uses the PHA commands to put these on the stack. Thus when the RTS is encountered, the program returns to DONE. Notice that we have seemingly violated two general rules of machine language programming. The first is using the PHA commands without corresponding PLA statements, and the second is the use of an RTS without a calling JSR.

Upon further thought however, it should become apparent that the two counteracted each other, and that an RTS is really equivalent to two PLAs.

The converse of this is when using two PLAs within a routine called by a JSR to avoid returning to the calling address. This is equivalent to using a POP command in an Applesoft subroutine called by a GOSUB.

Having thus simulated the JSR command, let's put it all together into a rewrite of the Print 1 routine that uses calls to subroutines to minimize the extra code required to make the routine relocatable:

```

1 *****
2 * RELOCATABLE PRINT 2 *
3 *****

```

				4 *
				5 OBJ \$300
				6 ORG \$300
				7 *
				8 PTR EQU \$06
				9 COUT EQU \$FDED
				10 RTRN EQU \$FF58
				11 STCK EQU \$100
				12 *
				13 *
0300:	20	58	FF	14 ENTRY JSR RTRN
0303:	B8			15 CLV
0304:	50	15		16 BVC PRINT
				17 *
0306:	D4	C5	D3	18 DATA1 ASC "TEST1"
030B:	8D	00		19 HEX 8D00
				20 *
030D:	20	58	FF	21 L2 JSR RTRN
0310:	B8			22 CLV
0311:	50	08		23 BVC PRINT
				24 *
0313:	D4	C5	D3	25 DATA2 ASC "TEST2"
0318:	8D	00		26 HEX 8D00
				27 *
031A:	60			28 DONE RTS
				29 *
031B:	BA			30 PRINT TSX
031C:	CA			31 DEX
031D:	18			32 CLC
031E:	BD	00	01	33 LDA STCK,X
0321:	69	04		34 ADC #\$04
0323:	85	06		35 STA PTR
0325:	E8			36 INX
0326:	BD	00	01	37 LDA STCK,X
0329:	69	00		38 ADC #\$00
032B:	85	07		39 STA PTR + 1
				40 *
032D:	A0	00		41 PRNT LDY #\$00
032F:	B1	06		42 LOOP LDA (PTR),Y
0331:	F0	06		43 BEQ FIX
0333:	20	ED	FD	44 JSR COUT
0336:	C8			45 INY
0337:	D0	F6		46 BNE LOOP ; (ALWAYS 'TILL 255)
				47 *
0339:	18			48 FIX CLC
033A:	98			49 TYA
033B:	65	06		50 ADC PTR
033D:	85	06		51 STA PTR
033F:	A5	07		52 LDA PTR + 1
0341:	69	00		53 ADC #\$00
0343:	48			54 PHA

```

0344:  A5 06      55  LDA PTR
0346:  48          56  PHA
0347:  60          57  FIN RTS
          58  *
          59  * WILL RTS TO L2/DONE
          60  *

```

This routine has the advantage of allowing the print statements to be used very much like they were in the nonrelocatable version given in chapter twelve. The extra bytes required for the stack calculations are confined to one place, and there are only three extra bytes per line to be printed, compared to the chapter twelve routine.

The return to the end of each printed string is accomplished by using the Y-Register in `FIX`. At entry to `FIX`, the Y-Register will hold the length of the string printed, which is then added to `PTR` to calculate the proper address to return to. Again we use the two `PHAS` followed by an `RTS` to accomplish the return.

Self-Modifying Code

Ah, here is an area to make the strongest heart quiver—the idea that a program rewrite itself to accomplish its given task. The possibilities are endless, but for now, we'll just look at a way of coping with statements like `LDA $ADDR,X`. It was this type of statement in the very first program of this chapter that contributed to its nonrelocatability. Here's the new mystery program:

```

          1  *****
          2  * RELOCATABLE PRINT 3 *
          3  *****
          4  *
          5  OBJ $300
          6  ORG $300
          7  *
          8  PTR EQU $06
          9  COUT EQU $FDED
         10  RTRN EQU $FF58
         11  STCK EQU $100
         12  *
         13  *
0300:    20 58 FF  14  ENTRY JSR RTRN
0303:    B8      15  CLV
0304:    50 14    16  BVC PRINT
          17  *
0306:    D4 C5 D3 18  DATA ASC "TEST"
030A:    8D 00    19  HEX 8D00

```

```

20 *
030C:  A2 00      21 PRNT LDY # $00
030E:  BD 06 03  22 LOOP LDA DATA,X
0311:  F0 06      23 BEQ DONE
0313:  20 ED FD   24 JSR COUT
0316:  E8         25 INX
0317:  D0 F5      26 BNE LOOP ; (ALWAYS TILL 255)
27 *
0319:  60         28 DONE RTS
29 *
031A:  BA         30 PRINT TSX
031B:  CA         31 DEX
031C:  18         32 CLC
031D:  BD 00 01  33 LDA STCK,X
0320:  69 04      34 ADC # $04
0322:  85 06      35 STA PTR
0324:  E8         36 INX
0325:  BD 00 01  37 LDA STCK,X
0328:  69 00      38 ADC # $00
032A:  85 07      39 STA PTR + 1
40 *
41 *
032C:  A0 09      42 FIX LDY # $09 ; LEN OF $ + 5
032E:  A5 06      43 LDA PTR
0330:  91 06      44 STA (PTR),Y
0332:  C8         45 INY
0333:  A5 07      46 LDA PTR + 1
0335:  91 06      47 STA (PTR),Y ; REWRITE DATA ADDR
0337:  B8         48 CLV
0338:  50 D2      49 BCV PRNT
50 *

```

This program will actually rewrite the address present on line 22 for the LDA DATA,X statement. The method uses the address on the stack to calculate the address for the beginning of the ASCII string to be printed. It is this address that we will want to eventually put into the code at \$30F,\$310 to rewrite the data statement.

After calculating the address in lines 30–39, the result is stored in PTR. The FIX section then adds the length of the printed string, plus five and uses this as the Y-Register offset to finally point to \$30F. The low and high-order bytes are then written to the code and a return done to the actual PRNT routine.

The example comes with many cautions. The value on line 42 must be appropriate to the length of the string being printed. Also the order of the ENTRY, DATA, and PRNT routines was deliberately chosen to make the rewrite as easy as possible. Extreme

care must be taken whenever constructing a program that alters itself, but the results can be very powerful.

If you are inclined to pursue this, study this example well until you are very sure why each step was done. To verify its versatility, you should assemble the code for this example and then run it at several different memory locations. After each run, list the code from the Monitor and see how the statement on line 22 has been rewritten. It's really quite fascinating!

Indirect Jumps

To round out this chapter, one more technique will be discussed. Although the stepping method using the forced branching can be used to span large distances, it can get rather inconvenient to have to keep inserting the stepping points throughout your code. An alternate technique is to use the indirect JMP command.

In the indirect jump, a two-byte pointer is created which indicates where the jump should be made to. The added advantage of this command is that the pointer need not be created on the zero page, which already is in high demand for numerous other uses. The basic syntax for the indirect jump is:

```
      .
      .
      .
0300: 6C FF FF 99 JI JMP ($FFF)
      .
      .
      .
```

Here is a sample program showing how this can be combined with the stack operation to create a relocatable jump command:

```
1 *****
2 * RELOCATABLE JMP SIM *
3 *****
4 *
5   OBJ $300
6   ORG $300
7 *
8   PTR EQU $06
9   BELL EQU $FF3A
10  RTRN EQU $FF58
11  STCK EQU $100
12 *
```

```

13 *
0300: 20 58 FF 14 ENTRY JSR RTRN
15 *
0303: BA 16 CALC TSX
0304: CA 17 DEX
0305: 18 18 CLC
0306: BD 00 01 19 LDA STCK,X
0309: 69 17 20 ADC #$17
030B: 85 06 21 STA PTR
030D: E8 22 INX
030E: BD 00 01 23 LDA STCK,X
0311: 69 00 24 ADC #$00
0313: 85 07 25 STA PTR+1
0315: 6C 06 00 26 JMP (PTR) ; TO 'EXPT'
27 *
0318: EA 28 FILI NOP
29 *
0319: 20 3A FF 30 EXPT JSR BELL
31 *
031C: 60 32 DONE RTS

```

The system is fairly simple, basically just using the stack to get a base address, and then adding whatever the distance is between the end of the JSR RTRN statement and the destination of the JMP(). As with some of the other systems though, this distance will change as code is added or deleted between the two points. Consequently you may have to change the values on lines 20 and 24 rather frequently to keep up with your code changes.

It does however avoid the problems associated with many stepping points sprinkled throughout your code, as would be necessary using the other alternative.

There is one bug in the use of the indirect jump that should be mentioned. It is present in the 6502 microprocessor itself, and occurs whenever the indirect pointer straddles a page boundary. For example, if you used the statement JMP (\$06), the destination would be retrieved from locations \$06 and \$07. However, if you were to use JMP (\$3FF), the destination would be retrieved from \$3FF and \$300. The high-order byte is not properly incremented by the 6502. This is usually not a concern, though, since there are generally many alternate locations for the destination pointer.

In conclusion then, certain techniques can be used to produce code which is not restricted to running at a particular address in memory. Although a bit harder to construct initially, and slightly larger in terms of final memory requirements, the final product is generally much more versatile in its applications.

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

1000

APPENDIX A

Assembly Lines Contest

In the March 1981 edition of *Softalk* magazine, Roger challenged the readers of his Assembly Lines column to a contest. Using the commands discussed in the column from October 1980 through March 1981 (all material covered through chapter five in this book). Contestants were asked to submit programs which would be judged by Roger, the shortest and most interesting program being the winner. Contest rules are reprinted here as they originally appeared in the March issue of *Softalk*.

Contest Rules

Create the shortest possible program using all and only the commands presented thus far in this series that does something interesting. The program must be entirely in machine language, and may not call any routines in Integer or Applesoft. It may call any of the Monitor routines from \$F000-\$FFFF.

The person who submits the shortest program of the most interest will be awarded \$50 worth of product from any advertiser in this issue of *Softalk* and the program will be published in *Softalk*.

Judging will be based on the opinions of a rather subjectively selected panel made up of people at *Softalk*, myself, and any other hapless passerby we can rope into this thing. Members of the staffs of *Softalk* and *Southwestern Data Systems* and professional programmers are not eligible to win. Entries should be submitted no later than April 15, 1981. Ties will be settled by Apple's random number generator. (I promise not to seed it!)

Contest results were announced in the June 1981 edition of *Softalk*. The winning program for Roger's contest is listed below. Roger's commentary accompanies the listing.

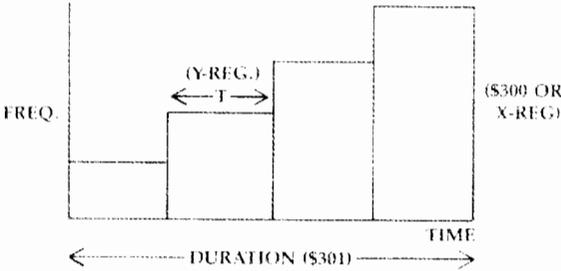
Contest Results

With the usual comments in mind about how hard it was to

decide on a winner, I hereby announce the winner of the contest as Steven Morris, of Queens, New York. His program combines a number of the principles we've discussed so far and also shows some nice touches in programming. It's an elegant use of all the given codes. Of particular interest is a self-modifying part wherein the program actually rewrites a small portion of itself upon user command.

I think it will be of interest, and also a good review, to go through Morris's listing to see what's been done. Before doing that, however, a little background on one more kind of tone routine is in order. This will make Morris's program that much more understandable.

In chapter seven, I discussed simple tone routines in which the speaker was accessed at a constant rate for a given length of time. These two factors determined the pitch and duration of the tone played. A variation on this is to have the pitch decrease or increase as the tone is played, creating effects rather like the sound usually associated with a falling bomb or a rising siren, respectively. This requires three variables, and without getting too technical, let me take a moment to illustrate with this chart:



The vertical axis represents the frequency of the tone being played. Putting several tones together into a series over a period of time creates, in this case, a rising scale. As each tone is played, the pitch is increased. Each individual tone lasts some arbitrary time, *T*, and put together, the series lasts an overall time period, labeled here as *Duration*.

If the pitch is decreased by a certain amount each time, the pattern is reversed. This is sometimes called a *ramp* tone pattern. In parentheses, I have indicated how each of these values is determined in Morris's program.

Here is a source listing of the program:

```

1 *****
2 * By Steven Morris *
3 *****
4 *
5   OBJ $302
6   ORG $302
7 *
8 PTCH EQU $300
9 DRTN EQU $301
10 SPKR EQU $C030
11 PREAD EQU $FB1E
12 PB0 EQU $C061
13 PB1 EQU $C062
14 GRSW EQU $C050
15 TXTSW EQU $C051
16 CLRSCR EQU $F832
17 *
18 LOOP DEX           ; DEC THIS DELAY
19   BNE CYCLE       ; DONE? NO = SKIP CLK
20 *
21 CLK LDX PTCH      ; REFRESH X-REG
22   LDA SPKR        ; CLK SPKR
23 * SPKR CLKS ONLY ONCE
24 * FOR EVERY ($300) PASSES
25 *
26 CYCLE DEY         ; # OF CYCLE CTR.
27   BNE LOOP        ; DONE?
28 *                 NO = KEEP GOING
29   DEC DRTN
30   BEQ CHPDL       ; DONE W/ RAMP?
31 *                 YES = CHK PDLs
32 RAMP INC PTCH
33   JMP LOOP
34 *
35 CHPDL LDX #$00
36   JSR PREAD        ; READ PDL (0)
37   STY PTCH         ; SET PTCH
38   INX
39   JSR PREAD        ; READ PDL (1)
40   STY DRTN         ; SET DRTN
41   LDY #$7F
42   CPY PB1          ; #1 PRESSED?
43   BCC TOGGLE       ; BRCH IF YES
44 *
45   INY              ; #$7F-->#$80 ; AN EXCUSE
46   TYA              ; TO USE THESE
47   TAX              ; COMMANDS.
48   CPX PB0          ; #0 PRESSED?
49   BCS LOOP         ; BRCH IF NO
50 *

```

```

51 SCREEN JSR CLRSCR ; CLR TOBLK
52 S1 STA GRSW ; SHOW GRAPHICS MODE
53 STA TXTSW ; SHOW TEXT MODE
54 JMP S1
55 *
56 SETDEC TAY ; USE UP THIS CODE
57 LDX # $CE ; OPCODE FOR 'DEC'
58 TXA
59 CMP RAMP ; IS IT 'DEC' NOW?
60 BEQ SETINC ; BRCH IF YES.
61 STA RAMP ; NO. MAKE IT 'DEC'
62 RTS
63 *
64 SETINC LDX # $EE ; OPCODE FOR 'INC'
65 STX RAMP
66 RTS
67 *
68 TOGGLE JSR SETDEC
69 JMP LOOP
70 *

```

This lists in memory as:

*300L

```

0300- 38          SEC
0301- A5 CA      LDA      $CA
0303- D0 06      BNE      $030B
0305- AE 00 03   LDX      $0300
0308- AD 30 C0   LDA      $C030
030B- 88         DEY
030C- D0 F4      BNE      $0302
030E- CE 01 03   DEC      $0301
0311- F0 06      BEQ      $0319
0313- CE 00 03   DEC      $0300
0316- 4C 02 03   JMP      $0302
0319- A2 00      LDX      #$00
031B- 20 1E FB   JSR      $FB1E
031E- 8C 00 03   STY      $0300
0321- E8         INX
0322- 20 1E FB   JSR      $FB1E
0325- 8C 01 03   STY      $0301
0328- A0 7F      LDY      #$7F
032A- CC 62 C0   CPY      $C062
032D- 90 27      BCC      $0356
032F- C8         INY
0330- 98         TYA
0331- AA         TAX
0332- EC 61 C0   CPX      $C061
0335- B0 CB      BCS      $0302

```

```

0337- 20 32 F8 JSR $F832
033A- 8D 50 C0 STA $C050
033D- 8D 51 C0 STA $C051
0340- 4C 3A 03 JMP $033A
0343- A8 TAY
0344- A2 CE LDX #$CE
0346- 8A TXA
0347- CD 13 03 CMP $0313
034A- F0 04 BEQ $0350
034C- 8D 13 03 STA $0313
034F- 60 RTS
0350- A2 EE LDX $EE
0352- 8E 13 03 STX $0313
0355- 60 RTS
0356- 20 43 03 JSR $0343
0359- 4C 02 03 JMP $0302

```

I'll try to explain each part of the program, hopefully with a proper balance of enough detail to jog your memory and enough brevity to keep things reasonably short.

If all this seems overwhelming, you're trying to read through it too fast. Go back through it slowly, taking your time. Have a nice cup of tea while you're at it.

Remember, we're packing seven chapters' worth of subject matter into one program. Don't worry if the fine details of the tone routine escape you. The important part is to make sure that you at least recall the existence and general nature of each individual command used in the program.

To explain the program, the easiest place to start is actually at `CHKPDL`, where the paddles are checked for new values at the end of each ramp series (line 35@319). The X-Register is loaded with a \$00 to tell the computer we want to read paddle zero in the next step, then `JSR` to `$FB1E`. That returns with the Y-Register holding the value of the paddle (\$00 to `$FF`), which is then stored in location \$300, labeled `PTCH` (for pitch). The X-Register value is then incremented from \$00 to \$01 on line 38, and paddle one read. The returned value is stored at \$301 for the duration value.

If paddle button one is pressed, location `$C062` will hold a number greater than `$7F`. To check for this, the Y-Register is loaded with `$7F` and compared against `$C062`. If `$C062` holds a value greater than `$7F`, the branch carry clear (`BCC`) will be taken (Y-Register < memory location = carry clear). We'll see what that does later.

If the value is less than `$7F`, program execution will fall

through to line 45. Here the \$7F is increased to \$80 and that value passed to the X-Register via the Accumulator. These steps are here to exercise the INY, TYA, TAX commands, and to allow us to use the CPX command next to fulfill the contest requirements. At line 48 the comparison is done. If the X-Register is greater (remember it holds a \$80 here), the button is not pressed and the branch carry set will be taken (X-Register > memory loc = carry set) that sends us to the main tone loop.

At entry to this loop, the X-Register and the Y-Register hold rather arbitrary values, but the overall theory is that, starting at CLK on line 21, the X-Register is loaded with the pitch value and the speaker clicked once. At line 26 the Y-Register is decremented; this is a counter for the length of that pitch value. Jumping back to loop, the net effect is that the program will make n passes through before clicking the speaker once, where n is the pitch value held in \$300. This creates the delay between clicks needed for a given tone.

The length of that particular tone is determined by the Y-Register. When it reaches a value of \$00, the BNE (branch not equal) fails and the counter for the overall duration is decremented. As long as there's time left (that is, DRIN > \$00), the next test fails (BEQ = branch if equal to zero) and the pitch value is incremented.

Going back to LOOP plays this next note until all the notes in the series have been played. Incrementing pitch gives a descending note pattern. (Recall that the greater the pitch value, the lower the tone played.)

When DRIN does reach zero, the program branches to the paddle check routine that we started in. Let's see what happens when a button is pressed. If button one is pressed, the program goes via TOGGLE to SETDEC. This clever section (ignore the TAY) loads the X-Register with the value \$CE. This is the opcode for DEC (decrement a memory location).

If the comparison fails, that is, there is not a \$CE currently there, the \$CE is stored at RAMP, the RTS (return from subroutine) returns to TOGGLE and the JMP LOOP sends everything back into the tone loop, this time with a DEC PITCH there instead. This gives an ascending pitch series.

If the comparison is true, it means that a \$CE was put there earlier, and the BEQ goes to SETINC, which restores the code for INC at RAMP (\$313), and then returns with the RTS, JMP LOOP as in the previous case.

These two options give the program the ability to rewrite itself, an interesting and powerful idea.

If paddle button zero is pressed, the branch at line 49 fails and the program falls into an infinite loop at SCREEN (\$337). In this loop, the screen is cleared to the color black by the monitor routine at \$F832.

Locations \$C050 and \$C051 are softswitches mentioned in earlier chapters. Remember that accessing these changes the display mode of the Apple. The screen can be viewed either in a text mode or in a graphics mode. Accessing \$C050 on line 52 sets the graphics mode, so the screen appears black. Accessing \$C051 sets the display to text, which appears as inverse "@" signs.

The JMP S1 repeats this cycle back and forth so fast that you don't actually see the flicker, just an interesting pattern created by the screens switching faster than your screen monitor can display them.

At this point you have to hit RESET to end.

There were a number of other excellent entries. Honorable mention should be made of Steve Hawley, Ray Ransom, Stephen Gagola, Jr., and Matt Brookover for their efforts.

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

8888

APPENDIX B

This section may well serve as the most often used portion of this book. I have mentioned elsewhere that learning programming can be looked upon as merely familiarizing yourself with the available tools to accomplish a specified task. The following section summarizes the tools available to a machine language programmer.

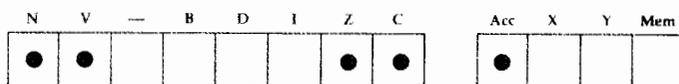
When you are first learning to program, much can be gained by simply browsing through the following pages and casually noting the variety of instructions available when writing a routine. Each entry provides not only the usual technical data on the instruction, but often a brief example of its use as well.

Please note that in some examples a percent sign (%) is used to indicate a binary form of a number. Some assemblers support this delimiter which can be very convenient, particularly when working with the logical operators and shift instructions. For example, the following representations are all equivalent:
 $100 = \$64 = \%01100100.$

ADC: Add with Carry

DESCRIPTION: This instruction adds the contents of a memory location or immediate value to the contents of the Accumulator, plus the carry bit, if it was set. The result is put back in the Accumulator. ADC works for both binary and BCD modes.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	ADC \$FFff	6D ff FF
Zero Page	ADC \$FF	65 FF
Immediate	ADC #\$FF	69 FF
Absolute,X	ADC \$FFff,X	7D ff FF
Absolute,Y	ADC \$FFff,Y	79 ff FF
(Indirect),X	ADC (\$FF,X)	61 FF
(Indirect),Y	ADC (\$FF),Y	71 FF
Zero Page,X	ADC \$FF,X	75 FF

USES: Peculiarly enough, ADC is most often used to add numbers together. Here are some common examples:

I. Adding a constant to a register or memory location:

```
CLC
LDA  MEM
ADC  #$80
STA  MEM
```

(MEM = MEM + #\$80)

II. Adding a constant (such as an offset) to a two-byte memory pointer:

```
CLC
LDA  MEM
ADC  #$80
STA  MEM
```

LDA MEM+1
ADC #\$80
STA MEM+1

(MEM, MEM+1 = MEM, MEM+1 + #\$80)

III. Adding two (2) two-byte values together:

CLC
LDA MEM
ADC MEM2
STA MEM
LDA MEM+1
ADC MEM2+1
STA MEM+1

(MEM, MEM+1 = MEM, MEM+1 + MEM2, MEM2+1)

AND: Logical AND

DESCRIPTION: This instruction takes each bit of the Accumulator and performs a logical AND with each corresponding bit of the specified memory location or immediate value. The result is put back in the Accumulator. The memory location specified is unaffected. (See ORA also.)

AND means that if *both* bits are one then the result will be one, otherwise the result will be zero.

The truth table used is:

	0	1
0	0	0
1	0	1

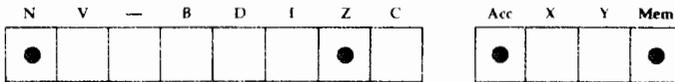
Example:

Accumulator: 0 0 1 1 0 0 1 1

Memory: 0 1 0 1 0 1 0 1

Result: 0 0 0 1 0 0 0 1

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	AND SFFff	2D ff FF
Zero Page	AND SFF	25 FF
Immediate	AND #\$FF	29 FF
Absolute,X	AND \$FFf,X	3D ff FF
Absolute,Y	AND SFFf,Y	39 ff FF
(Indirect,X)	AND (\$F,X)	21 FF
(Indirect),Y	AND (\$F),Y	31 FF
Zero Page,X	AND \$F,X	35 FF

USES: AND is used primarily as a *mask*, that is, to let only certain bit patterns through a section of a program. The mask is created by putting ones in each bit position where data is to be allowed through, and zeros where data is to be suppressed. For example, it is frequently desirable to mask out the high-order bit of ASCII data, such as would come from the keyboard or another input device (such as a disk file). The routine shown assures that no

matter what value is gotten from `DEVICE`, the high-order bit of the value put in `MEM` will always be clear:

<u>Routine</u>	<u>Sample input:</u>		
<code>LDA DEVICE</code>	<code>Accumulator:</code>	<code>01010111</code>	<code>11010111</code>
<code>AND #7F ; %01111111</code>	<code>#\$F</code>	<u><code>01111111</code></u>	<u><code>01111111</code></u>
<code>STA MEM</code>	<code>Result:</code>	<code>01010111</code>	<code>01010111</code>

`AND` is also used when you know the high bit will be set and you want it cleared. This is the case when getting ASCII characters from the keyboard. A common routine to get a character from the keyboard is:

```

WATCH   LDA   KYBD       ; $C000
         BPL   WATCH     ; AGAIN IF < #$80
         BIT   STROBE    ; CLEAR STROBE: $C010
         AND  #$7F      ; CLR HIGH BIT
         STA  MEM
    
```

Another way of looking at this same effect is to say that `AND` can be used to force a zero in any desired position in a byte's bit pattern. (See `ORA` to force ones). A zero is put in the mask value at the positions to be forced to zero, and all remaining positions are set to one. Whenever a data byte is `AND`ed with this mask, a zero will be forced at each position marked with a zero in the mask, while all other positions will be unaffected, remaining zeros or ones, as in their original condition.

There are also at least two other rather obscure uses for the `AND` instruction. The first of these is to do the equivalent of a `MOD` function, involving a piece of data and a power of two. You'll recall that the `MOD` function produces the *remainder* of a division operation. For example: $12 \text{ MOD } 4 = 0$; $14 \text{ MOD } 4 = 2$; $18 \text{ MOD } 4 = 2$; $17 \text{ MOD } 2 = 1$; etc.

The general formula is: $\text{Acc. MOD } 2^n = \text{RESULT}$

The actual operation is carried out by using a value of $(2^n - 1)$ as the mask value. The theory of operation is that only the last n bits of the data byte are let through, thus producing the result corresponding to a `MOD` function.

Example:

```

LDA   MEM
AND   #$07      ; %00000111 = 23 - 1
STA  MEM       ; MEM = MEM MOD 8
    
```

This technique provides one of several ways of testing for the odd/even attribute of a number:

```
LDA    MEM
AND    #$01      ; %00000001 = 21 - 1
BEQ    EVEN
BNE    ODD
```

The result of the AND of any number and #\$01 will always be either 0 or 1 depending on whether the number was odd or even.

The third application is in determining if a given bit pattern is present among the other data in a number. For example, to test if bits 0, 3 and 7 are on:

```
LDA    MEM
AND    #$89      ; %10001001
CMP    #$89
BEQ    MATCH
BNE    NOMATCH
```

The general technique is to first AND the data against the value for the byte with just the desired bits set to one (all others zero), and then immediately do a CMP to the same value. If all the specified bits match, a BEQ will succeed.

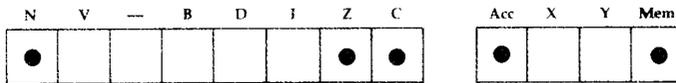
NOTE: BIT (described later) can be used to test for one or more matches, but the AND technique described here confirms that *all* the bits of interest match.

ASL: Arithmetic Shift Left

DESCRIPTION: This instruction moves each bit of the Accumulator or specified memory location one position to the left. A zero is forced at the bit 0 position, and bit 7 (the high-order bit) falls into the carry. The result is left in the Accumulator or memory location.

(See also ROL; also LSR and ROR.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Accumulator	ASL	0A
Absolute	ASL \$FFff	0E ff FF
Zero Page	ASL \$FF	06 ff FF
Absolute,X	ASL \$FFff	1E ff FF
Zero Page,X	ASL \$FE,X	16 FF

USES: The most common use of ASL is for multiplying by a power of two. You are already familiar with the effect in base ten: $123 * 10 = 1230$ (shift left). For example:

```

LDA    MEM
ASL                    ; TIMES 2
ASL                    ; TIMES 2 AGAIN
STA    MEM            ; MEM = MEM * 4 (4 = 22)

```

The other use is to check a given bit position by conditioning the carry flag. For example, to check bit 4, this would look like:

```

LDA    MEM
ASL
ASL
ASL                    ; BIT 4 NOW IN CARRY
BCS    SET
BCC    NOTSET

```

NOTE: This technique destroys the Accumulator in the process of checking the bit. AND or BIT instructions are generally preferred instead of this technique.

If testing bits 0-3, LSR or ROR may be more appropriate (fewer shifts needed). ROL can also be used instead of ASL depending on whether or not the data is to be preserved.

BCC: Branch Carry Clear

DESCRIPTION: Executes a branch if the carry flag is clear. Ignored if carry is set. Many assemblers have an equivalent pseudo-op called BLT (Branch Less Than, not to be confused with the sandwich), since BCC is often used immediately following a comparison to see if the Accumulator held a value less than the specified value.

FLAGS & REGISTERS AFFECTED:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem

ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BCC ADDRESS	6D FF

NOTE: The carry flag, upon which this depends, is conditioned by: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLP, ROL, RTI, SBC, and SEC.

USES: As mentioned, BCC is used to detect when the Accumulator is less than a specified value. The usual appearance of the code is listed below. Note that in a two-byte comparison the high-order bytes are checked first.

One-Byte Comparison:

ENTRY	LDA	MEM
	CMP	MEM2
	BCC	LESS
	BCS	EQ/GRTR

(Goes to LESS if MEM < MEM2)

Two-Byte Comparison:

ENTRY	LDA	MEM+1
	CMP	MEM2+1
	BCC	LESS ; MEM+1 < MEM2+1
	BEQ	CHK2 ; MEM+1 = MEM2+1
	BCS	GRTR ; MEM+1 > MEM2+1

```

CHK2   LDA   MEM
        CMP   MEM2
        BCC  LESS    ; MEM    <    MEM2
        BCS  EQ/GRTR ; MEM    <=   MEM2

```

(Goes to LESS only if $MEM, MEM+1 < MEM2, MEM2+1$)

DESCRIPTION: Executes the branch only if the carry flag is set. Some assemblers support the pseudo-op BGT (for Branch Greater Than), since this command is used to test for the Accumulator equal to or greater than the specified value.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BCS ADDRESS	B0 FF

NOTE: The carry flag, upon which this depends, is conditioned by: ADC, ASL, CLC, CMP, CPX, CPY, LSR, PLE, ROL, RFL, SBC, and SEC.

USES: Used to detect Accumulator equal to or greater than a specified value. Can be combined with BEQ to detect a greater than relationship. Note that in the two-byte comparison, the high-order bytes are checked first.

One-Byte Comparison:

ENTRY	LDA	MEM
	CMP	MEM2
	BCC	LESS
	BEQ	EQUAL
	BCS	GREATER

(Goes to GREATER if MEM > MEM2, or EQUAL if MEM = MEM2)

Two-Byte Comparison:

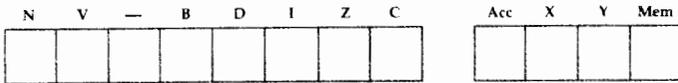
ENTRY	LDA	MEM+1			
	CMP	MEM2+1			
	BCC	LESS	; MEM+1	<	MEM2+1
	BEQ	CHK2	; MEM+1	=	MEM2+1
	BCS	GRTR	; MEM+1	>	MEM2+1
CHK2	LDA	MEM			
	CMP	MEM2			
	BCC	LESS	; MEM	<	MEM2
	BEQ	EQUAL	; MEM	=	MEM2
	BCS	GREATER	; MEM	>	MEM2

(Goes to GREATER only if MEM, MEM+1 > MEM2, MEM2+1, or to EQUAL if MEM, MEM+1 = MEM2, MEM2+1)

BEQ: Branch if Equal

DESCRIPTION: Executes a branch if the Z-flag (zero flag) is set, indicating that the result of a previous operation was zero. See BCS to see how a comparison for Accumulator equal to value is done.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BEQ ADDRESS	F0 FF

NOTE: The zero flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, RTS, SBC, TAX, TAY, TXA, and TYA.

USES: In addition to being used in conjunction with compare operations, BEQ is used to test for whether the result of a variety of other operations has resulted in a value of zero. The common classes of these operations are increment/decrement, logical operators, shifts, and register loads. Even easier to remember is the general principle that whenever you've done something that results in zero, chances are good the Z-flag has been set. Likewise, any non-zero result of an operation is likely to clear the Z-flag. One of the most common instances is when checking an input string for a zero, usually used as a delimiter:

Example:

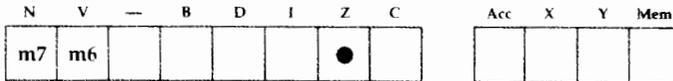
```
ENTRY   LDA   DEVICE
        BEQ   DONE ; DATA = 0
WORK    (...)
        JMP   ENTRY
DONE    RTS
```

BIT: Compare Accumulator BITS with contents of memory

DESCRIPTION: Performs a logical AND on the bits of the Accumulator and the contents of the memory location. The opposite of the result is stored in the Z-flag. What this means is that if any bits set in the Accumulator happen to match any set in the value specified, the Z-flag will be cleared. If no match is found, it will be set. BNE is used to detect a match, BEQ detects a no match condition.

Fully understanding the function and various applications of this instruction is a sign of *having arrived* as a machine language programmer, and suggests you are probably the hit of parties, thrilling your friends by doing hex arithmetic in your head and reciting ASCII codes on command.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING		
Absolute	BIT \$FFFF	2C	ff	FF
Zero Page	BIT \$FF	24	FF	

*Note absence of the immediate mode as an option!

USES: BIT provides a means of testing whether or not a given bit is *on* in a byte of data. IMPORTANT: BIT will only indicate that at least one of the bits in question match. It does not indicate how many actually do match. See the AND instruction on how to do a check for all matching.

The test mask can either be held in the Accumulator (if testing a memory location), or in a memory location (when testing the Accumulator). The mask is created by setting ones in the bit positions you are interested in, and leaving all remaining positions set to zero.

Examples:

I. Showing the results of the BIT operation:

Acc: 1 0 0 1 1 0 1 1
Mem: 0 1 0 1 0 1 0 1
Result: 0 0 0 1 0 0 0 1 → 1 → (opposite) → 0

Z-flag effect:
BNE works
BEQ not taken

STATUS REGISTER:

N	V	—	B	D	I	Z	C
0	1					0	

II.

Acc: 1 0 0 1 1 0 1 1
Mem: 0 1 0 0 0 1 0 0
Result: 0 0 0 0 0 0 0 0 → 0 → (opposite) → 1

Z-flag effect:
BEQ works
BNE not taken

STATUS REGISTER:

N	V	—	B	D	I	Z	C
0	1					1	

III. Sample routines.

Test Acc. for bit 4 on

```
ENTRY   LDA   #$10           ; %00010000
        STA   MEM
        LDA   DEVICE
        BIT   MEM
        BNE   MATCH
        BEQ   NOMATCH
```

Test memory for bit 4 on

```
ENTRY   LDA   #$10           ; %00010000
        BIT   MEM
        BNE   MATCH
        BEQ   NOMATCH
```

BIT also sets the N and V-flags, and thus provides a very fast way of testing bits 6 and 7. Since bit 7 is the high-order bit and is frequently used to indicate certain conditions, this can be quite handy. Here is an example on how to watch for a keypress:

```

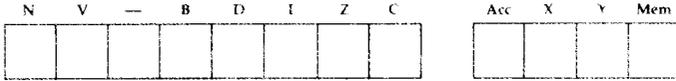
LOOP   BIT   KYBD       ; %C000
      BPL   LOOP       ; VAL < 128 = NOT PRESS
      BIT   STROBE     ; %C010
DONE   RTS

```

Note that in this example, no data is actually retrieved from the keyboard. Only a wait is done until the keypress. The BIT STROBE step in the example also provides an illustration of a second application of BIT, which is to access a hardware location (often called a *softswitch*) without damaging the contents of the Accumulator.

DESCRIPTION: Executes branch only if N-flag (sign flag) is set. N-flag is set by any operation producing a result in the range \$80 to \$FF (i.e. high bit set).

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BMI ADDRESS	30 FF

NOTE: The zero flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, TAX, TAY, FXS, TXA, and TYA.

USES: BMI is most commonly used to detect negative numbers when signed binary math is used, but is also equally common in testing for a high bit set, such as in watching the keyboard for a keypress. (See BIT also.) For example:

```

LOOP   LDA   KYBD
        BMI  PRESS   ; DATA > $7F
        BPL  LOOP    ; DATA < $80
    
```

BMI is also useful for terminating a loop that you want to reach zero, and where the loop will otherwise stay out of the \$80 to \$FF range:

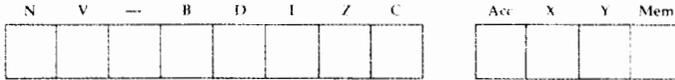
```

ENTRY  LDX   $20      ; TO LOOP 33 TIMES
LOOP   DEX
        BMI  DONE     ; WHEN X = $FF
        BPL  LOOP     ; WHILE X > $FF
DONE   RTS
    
```

BNE: Branch Not Equal

DESCRIPTION: Executes the branch if the Z-flag (zero flag) is clear, that is to say, if the result of an operation was a non-zero value.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BNE ADDRESS	D0 FF

NOTE: The zero flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, RTS, SBC, TAX, TAY, TXA, and TYA.

USES: Often used in loops to branch until the counter reaches zero. Also used in data input loops to verify the non-zero nature of the last byte in, as when checking for the end of data delimiter.

I. Simple loop

```
ENTRY   LDX   #$20           ; WILL COUNT 32 TIMES
LOOP    DEX
        BNE   LOOP          ; 'TILL X = 0
DONE    RTS
```

II. Data input routine

```
ENTRY   LDA   DEVICE
        BNE   CONTINUE
DONE    RTS
```

III. As used in a two-byte increment routine

```
ENTRY   LDA   MEM
        ADC   #$01
        STA   MEM
```

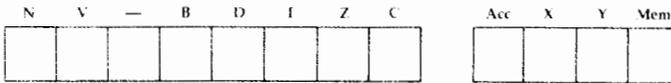
```

BNE    DONE          ;  UNLESS MEM = 0
LDA    MEM+1
ADC    #$00          ;  MEM+1 = MEM+1 + 1
STA    MEM+1
DONE   RTS
```

BPL: Branch on PLUS

DESCRIPTION: Executes branch only if N-flag (sign flag) is clear, as would be the case when the result of an operation is in the range of \$00 to \$7F (high bit clear). See also BMI.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BPL ADDRESS	10 FF

NOTE: The sign flag, upon which this depends, is conditioned by: ADC, AND, ASL, BIT, CMP, CPY, CPX, DEC, DEX, DEY, EOR, INC, INX, INY, LDA, LDX, LDY, LSR, ORA, PLA, PLP, ROL, TAX, TAY, TXS, TXA, and TYA.

USES: BPL is an easy way of staying in a loop until the high bit is set. It is also used in general to detect the status of the high bit. Here's our familiar keypress check using BPL:

```
ENTRY   LDA    KYBD      ; $C000
        BPL    ENTRY     ; LOOP UNTIL DATA > $7F
        BIT    STROBE    ; CLR $C010
        STA    MEM       ; SAVE VALUE
DONE    RTS
```

Also used for short loops that you want to reach zero:

```
ENTRY   LDX    #$20      ; WILL LOOP 33 TIMES
LOOP    DEX                    ; X = X - 1
        BPL    LOOP       ; UNTIL X = $FF
DONE    RTS
```

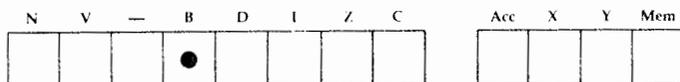
BRK: BReAK (software interrupt)

DESCRIPTION: When a BRK is encountered in a program, program execution halts, and the user generally sees something like the following:

```
0302- A = A0 X = 00 Y = 01 P = 36 S = E7
```

What actually happens is that the program counter, plus two, is saved on the stack, immediately followed by the Status Register, in which the BRK bit has been set. The processor then jumps to the address at \$FFFE,FFFF. On the Apple, this is a vector at \$3F0,3F1 which points to the BRK handler routine which produces the results mentioned above.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

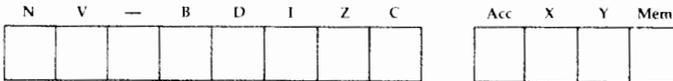
MODE	COMMON SYNTAX	HEX CODING
Implied Only	BRK	00

USES: BRK can be very useful in debugging machine language programs. A BRK is simply inserted into the code at strategic points in the routine. When the program comes to a screeching halt, one can examine the status of various memory locations and registers to see if everything is as you think it should be. This process can be formalized, and hence considerably improved on, by using a software utility called a debugger, which allows you to step through a program one instruction at a time. *Munch-A-Bug*, along with others, provides this option. On Integer Apples, a primitive Step and Trace function is provided as part of the Monitor.

BVC: Branch on oVerflow Clear

DESCRIPTION: Executes a branch only if the V-flag (overflow flag) is clear. The overflow flag is cleared whenever the result of an operation did not entail the carry of a bit from position 6 to position 7. The overflow flag can also be cleared with a CLV command.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BVC ADDRESS	50 FF

NOTE: The overflow flag, upon which this depends, is conditioned by: ADC, BIT, CLV, PLP, RFL, and SBC.

USES: BVC is used primarily in detecting a possible overflow from the data portion of the byte into the sign bit, when using signed binary numbers. For example:

ENTRY	CLC		
	LDA	#\$64	; %01100100 = + 100
	ADC	#\$40	; %01000000 = + 64
	BVC	STORE	; NOT TAKEN HERE
ERR	RTS		; RESULT = +164 =
			; %10100100 > \$7F
STORE	STA	MEM	

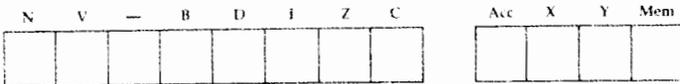
BVC can also be used as a forced branch when writing relocatable code. The advantage is that the carry remains unaffected, thus allowing it to be tested later in the conventional manner.

```
CLV          ; CLEAR V FLAG
BVC LABEL   ; (ALWAYS)
```

BVS: Branch overflow Set

DESCRIPTION: Executes the branch only when the V-flag (overflow flag) is set. The overflow flag is set only when the result of an operation causes a carry of a bit from position 6 to position 7. Note that there is not a command to specifically set the overflow flag (as would correspond to a SEC command for the carry).

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Relative Only	BVS ADDRESS	70 FF

NOTE: The overflow flag, upon which this depends, is conditioned by: ADC, BIT, CLV, PLE, RFI, and SBC.

USES: BVS is used primarily in detecting a possible overflow from the data portion of the byte into the sign bit, when using signed binary numbers. For example:

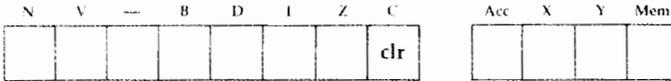
```
ENTRY    CLC
          LDA    #564      ; %01100100 = +100
          ADC    #$40      ; %01000000 = + 64
          BVS    ERR       ; RESULT = +164 =
                           ; %10100100 > $7F

STORE    STA    MEM
DONE     RTS
ERR      JSR    BELL      ; ALERT TO OVERFLOW
```

CLC: CLear Carry

DESCRIPTION: Clears the carry bit of the status register.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only	CLC	18

USES: CLC is usually required before the first ADC instruction of an addition operation, to make sure the carry hasn't inadvertently been set somewhere else in the program, and thus incorrectly added to the values used in the routine itself. A CLC can also be used to force a branch when writing relocatable code, such as:

```
CLC  
BCC LABEL ; (ALWAYS)
```

CLD: CLear Decimal Mode

DESCRIPTION: CLD is used to enter the binary mode (which the Apple is usually in by default), so as to properly use the ADC and SBC instructions. (See SED for setting decimal mode.)

FLAGS & REGISTERS AFFECTED:

N	V	—	B	D	I	Z	C	Acc	X	Y	Mem
				clr							

ADDRESSING MODES AVAILABLE:

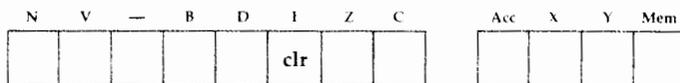
MODE	COMMON SYNTAX	HEX CODING
Implied Only	CLD	D8

USES: The arithmetic mode of the 6502 is an important point to keep in mind when using the ADC and SBC instructions. If you are in the wrong mode from what you might assume, rather unpredictable results can occur. See the SED instruction entry for more details on the other mode.

CLI: CLear Interrupt Mask

DESCRIPTION: This instruction enables interrupts.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

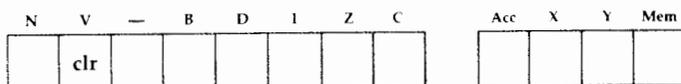
MODE	COMMON SYNTAX	HEX CODING
Implied Only	CLI	58

USES: CLI tells the 6502 to recognize incoming IRQ (Interrupt ReQuest) signals. The Apple's default is to have interrupts enabled, but after the first interrupt, all succeeding interrupts are disabled by the 6502 until a CLI is re-issued. As a matter of interest, timing dependent routines like the DOS RWTS (Read-Write Track Sector) routine disable interrupts while *on*, and then allow them again with the CLI at exit.

CLV: CLear oVerflow Flag

DESCRIPTION: This clears the overflow flag by setting the V bit of the Status Register to zero.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE

MODE	COMMON SYNTAX	HEX CODING
Implied Only	CLV	B8

USES: Because the overflow flag is, in fact, cleared by a non-overflow result of an ADC instruction, it is not usually necessary to clear the flag prior to an addition. It is, however, occasionally used as a relatively unobtrusive way of forcing a branch when writing relocatable code.

This is done in a manner similar to the CLC, BCC or SEC, BCS pairs discussed in chapter fourteen. The general form is:

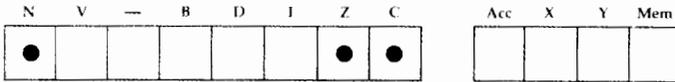
CLV
BVC ADDRESS

This technique has the advantage of not affecting the carry flag, should the user want to test the carry after the forced break.

CMP: CoMPare to Accumulator

DESCRIPTION: CMP compares the Accumulator to a specified value or memory location. The N (sign), Z (zero), and C (carry) flags are conditioned. A conditional branch is usually then done to determine whether the Accumulator was less than, equal to, or greater than the data.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE

MODE	COMMON SYNTAX	HEX CODING		
Absolute	CMP \$FFff	CD	ff	FF
Zero Page	CMP \$FF	C5	FF	
Immediate	CMP #\$FF	C9	FF	
Absolute,X	CMP \$FFff,X	DD	ff	FF
Absolute,Y	CMP \$FFff,Y	D9	ff	FF
(Indirect),X	CMP (\$FF,X)	C1	FF	
(Indirect),Y	CMP (\$FF),Y	D1	FF	
Zero Page,X	CMP \$FF,X	D5	FF	

USES: CMP is used to check the value of a byte against certain values such as would be done in loops, or in data processing routines. The routine typically decides whether the result is less than, equal to, or greater than a critical value. The usual pattern is:

```

BCC:   Acc.   <  value
BCS:   Acc.   >= value
BEQ, BCS: Acc. >  value
    
```

See the section on BCC through BCS for specific examples.

IMPORTANT: A `CMP #$00` should never be done. Consider this example:

```
ENTRY  LDY    #$FF
```

```

LOOP    DEY
        CPY    #$00
        BCS    LOOP    ; (ALWAYS TAKEN!)
        BCC    DONE
DONE    RTS

```

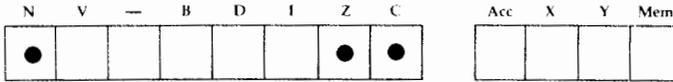
Because \$01 through \$FF is *larger* than zero, the branch will be taken while the Y-Register is in this range. Since \$0 = \$0, when Y reaches zero, the branch will still be taken. Therefore, the example creates an endless loop which will never terminate.

Similarly, if a BCC is done first, it will *never* be taken, since there is no value *less* than zero to trigger it.

CPX: ComPare data to the X-Register.

DESCRIPTION: CPX compares the contents of the X-Register against a specified value or memory location. See also CMP.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	CPX \$FFff	EC ff FF
Zero Page	CPX \$FF	E4 FF
Immediate	CPX #\$FF	E0 FF

USES: CPX is primarily used in loops which read data tables, with the X-Register being used as the offset in the Absolute,X addressing mode. The X-Register is usually loaded with zero, and then incremented until it reaches the length of the data stream to be read. For example:

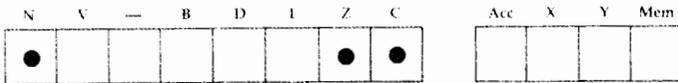
```
ENTRY  LDY    #$00
LOOP   LDA    DATA,X
        JSR    PRINT
        INX
        CPX    #$05
        BCC   LOOP
DONE   RTS
DATA   ASC    "TEST!"
```

For the same reasons discussed under CMP, a CPX #\$00 should not be used. See CMP for details.

CPY: ComPare data to the Y-Register

DESCRIPTION: CPY compares the contents of the Y-Register against a specified value or memory location. See also CMP.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	CPY \$FFFF	CC ff FF
Zero Page	CPY \$FF	C4 FF
Immediate	CPY #\$FF	C0 FF

USES: The Y-Register is usually used when reading a stream of data from a zero page pointer. CPY allows for checking the current value of the Y-Register against a critical value. In this example, the Y-Register is used to retrieve the first five bytes of an Apple-soft program line:

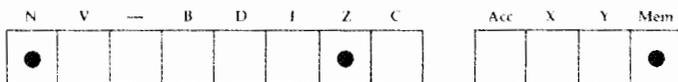
```
ENTRY  LDY  #$00
LOOP   LDA  ($67),Y    ; PROG BEG + Y
        STA  ($06),Y    ; TEMP STORAGE AREA
        INY
        CPY
        BCC  LOOP      ; LOOP FOR 5 BYTES
DONE   RTS
```

For the same reasons discussed under CMP, a CPY #\$00 should not be used. See CMP for details.

DEC: DECrement a memory location

DESCRIPTION: The contents of the specified memory location are decremented by one. If the original contents were equal to # $\$00$, then the result will *wrap around* giving a result of # $\$FF$.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	DEC \$FFff	CE ff FF
Zero Page	DEC \$FF	C6 FF FF
Absolute,X	DEC \$FFff,X	DE ff FF
Zero Page,X	DEC \$FF,X	D6 FF FF

USES: DEC is usually used when decrementing a one-byte memory value (such as an offset), or a two-byte memory pointer. Here are the common examples:

One Byte Value:

```
ENTRY   DEC   MEM
DONE    RTS
```

Two Byte Pointer:

```
ENTRY   DEC   MEM
        LDA   MEM
        CMP   #$FF      ; WRAP-AROUND?
        BNE   DONE      ; NO
        DEC   MEM+1     ; YES: DEC MEM+1
DONE    RTS
```

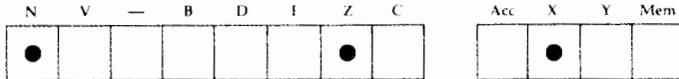
After the DEC operation, the N and/or Z-flags are often checked to see if the result was negative or a zero/non-zero value, respectively.

The technique shown for the two-byte decrement operation is not necessarily the most efficient. See the SBC entry for an alternative method.

DEX: DEcrement the X-Register

DESCRIPTION: The X-Register is decremented by one. When the original value was #00, the result will *wrap around* to give a result of #FF. See also DEC.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	DEX	CA

USES: DEX is often used in reading a data block via indexed addressing, i.e. Absolute,X. Here is a simple example:

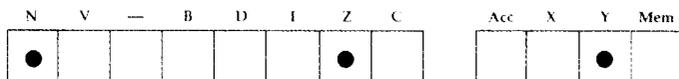
```
ENTRY   LDX   #$05
LOOP    LDA   DATA-1,X
        JSR   PRINT
        DEX
        BNE  LOOP
DONE    RTS
DATA    ASC   "ITSET"
```

NOTE: There are several points of interest in this example. Besides the general use of the X-Register in the indexed addressing mode, notice that the loop runs *backwards* from #05 to #01. The loop is terminated when the X-Register reaches zero. Because the loop runs from high memory down, the ASCII string is put in memory in reverse order, as evidenced in the listing. Also note that the base address of the loop is DATA-1. This allows the use of the #05 to #01 values of the X-Register.

DEY: DEcrement the Y-Register

DESCRIPTION: The Y-Register is decremented by one. When the original value was # $\$00$, the result will *wrap around* to give a result of # $\$FF$. See also DEC.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	DEY	88

USES: DEY is usually used when decrementing a reverse scan of a data block, using a zero page pointer via indirect indexed addressing (such as LDA ($\$FF$), Y). Reverse scans are often used because it's so easy to use a BEQ instruction to detect when you're done. DEY is also used when making a counter for a small number of cycles. Here's a routine which outputs a variable number of carriage returns, as indicated by the contents of MEM.

ENTRY	LDY	MEM	
LOOP	LDA	# $\$8D$; <RETURN>
	JSR	COUT	; $\$FDED$
	DEY		
	BNE	LOOP	; 'TILL Y=0
DONE	RTS		

EOR: Exclusive-OR with Accumulator

DESCRIPTION: The contents of the Accumulator is Exclusive-ORed with the specified data. The N (sign) and Z (zero) flags are also conditioned depending on the result. The result is put back in the Accumulator. The memory location (if specified) is unaffected.

EOR means that if either bit, but *not both*, is one then the result will be one, otherwise the result will be zero.

The truth table used is:

	0	1
0	0	1
1	1	0

Example:

Accumulator:

0 0 1 1 0 0 1 1

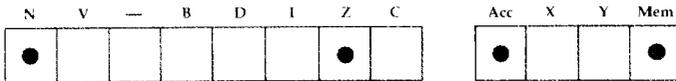
Memory:

0 1 0 1 0 1 0 1

Result:

0 1 1 0 0 1 1 0

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	EOR \$FFff	4D ff FF
Zero Page	EOR \$FF	45 FF
Immediate	EOR #\$FF	49 FF
Absolute,X	EOR \$FFff,X	5D ff FF
Absolute,Y	EOR \$FFff,Y	59 ff FF
(Indirect),X	EOR (\$F,X)	41 FF
(Indirect),Y	EOR (\$F),Y	51 FF
Zero Page,X	EOR \$FF,X	55 FF

USES: EOR has a wide variety of uses. (1) The most common is to encode data by doing an EOR with an arbitrary one-byte *key*. The data may then be decoded later by again doing an EOR of each data byte with the same key again.

```

CODE      LDX      #$05
LOOP      LDA      DATA1,X
          EOR      #$7D      ; ARBITRARY "KEY"
          STA      $300,X    ; REWRITE TABLE
          DEX
          BNE      LOOP      ; 'TILL X = 0
DONE      RTS
DATA      ASC      "TEST!"

DECODE    LDX      #$05
LOOP      LDA      $300,X    ; RETRIEVE CODED DATA
          EOR      #$7D
          STA      $380,X    ; PUT IN NEW LOC.
          DEX
          BNE      LOOP
DONE      RTS

```

(2) Another application is to reverse a given bit of a data byte. The mask is created by putting a one in the positions which you wish to have reversed. A zero is put in all remaining positions. When the EOR with the mask is done, bits in the specified positions will *reverse*, i.e. ones will become zeros, and vice versa. See the original example in this entry to verify this effect.

(3) The N (sign) flag can be used to detect if both memory and the Accumulator have bit 7 set:

```

ENTRY     LDA      MEM
          EOR      MEM2
          BMI      MATCH    ; BOTH SET
          BPL      NOMATCH  ; BOTH NOT SET

```

(4) The Z (Zero) flag will be set if either the Accumulator or memory or both equal zero:

```

ENTRY     LDA      MEM
          EOR      MEM2
          BEQ      ZERO     ; MEM = 0 and/or MEM2 = 0
          BNE      NOTZ    ; NEITHER MEM NOR MEM2 = 0

```

(5) EOR is also useful in producing the twos' complement of a number for use in signed binary arithmetic.

```

ENTRY   LDA   #$34      ; %00110100 = +52
        ; TO BE CVRTD TO -52
        EOR   #$FF      ; %11111111 = $FF
        ; RSLT = %11001011
        CLC
        ADC   #$01      ; RSLT = RSLT + 1
        ; = %11001100 = $CC
        STA   MEM      ; STORE RSLT
DONE    RTS

```

(5a) And to convert signed negative numbers back:

```

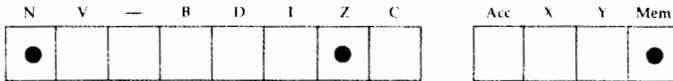
ENTRY   LDA   #$CC      ; %11001100 = $CC = -52
        ; TO BE CVRTD BACK
        SEC
        SBC   #$01      ; ACC = ACC - 1
        ; = %11001011 = $CB
        EOR   #$FF      ; REVERSE ALL BITS
        ; RSLT = %00110100 = $34 = +52
        STA   MEM      ; STORE RESULT
DONE    RTS

```

INC: INcrement Memory

DESCRIPTION: The contents of a specified memory location are incremented by one. If the original value was #\$FF, then incrementing will result in a *wrap around* giving a result of #\$00. The N (sign) and Z (zero) flags are conditioned depending on the result.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING		
Absolute	INC \$FFff	EE	ff	FF
Zero Page	INC \$FF	E6	FF	
Absolute,X	INC \$FFff,X	FE	ff	FF
Zero Page,X	INC \$FF,X	F6	FF	

USES: INC is most often used for incrementing a one-byte value (such as an offset) or a two-byte pointer. Here are the most common forms:

One-Byte Value

ENTRY	INC	MEM
	RTS	

Two-Byte Pointer

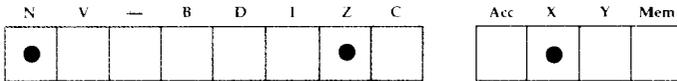
ENTRY	INC	MEM
	BNE	DONE
	INC	MEM + 1
	DONE	RTS

After the INC operation, the N and/or Z-flags are often checked to see if the result was negative or a zero/non-zero value, respectively.

INX: INcrement the X-Register

DESCRIPTION: The contents of the X-Register are incremented by one. If the original value was #\$FF, then incrementing will result in a *wrap around* giving a result of #\$00. The N (sign) and Z (zero) flags are conditioned depending on the result.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	INX	E8

USES: INX is used in forward scanning loops which digest a DATA stream as shown here:

```

ENTRY   LDX   #$00
LOOP    LDA   DATA,X
        BEQ   DONE      ; DELIMITER?
        JSR   COUT      ; $FDED
        JMP   LOOP     ; NEXT CHAR
DONE    RTS
DATA    ASC   "TEST!"
        HEX   00      ; END OF DATA
    
```

Note that in forward scanning loops, the base address can be DATA itself (See DEX for another situation).

INX can also be used as a general purpose counter for miscellaneous routines:

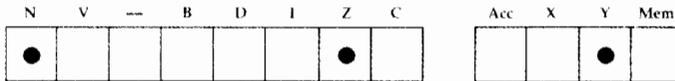
```

ENTRY   LDX   #$00
        LDA   #$8D      ; <RETURN>
LOOP    JSR   COUT      ; $FDED
        INX
        CPX   #$05
        BCC   LOOP     ; 'TILL X = 5
DONE    RTS           ; PRINTS 5 CR'S
    
```

INY: INcrement the Y-Register

DESCRIPTION: The contents of the Y-Register are incremented by one. If the original value was $\#\$FF$, then incrementing will result in a *wrap around* giving a result of $\#\$00$. The N (sign) and Z (zero) flags are conditioned depending on the result.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	INY	CB

USES: INY is used in forward scanning loops which use the indirect indexed addressing mode (ie. LDA ($\$FF$), Y). This is quite common in routines which process strings for certain characters, search routines, etc. Here is a routine which scans the input buffer for the first carriage return:

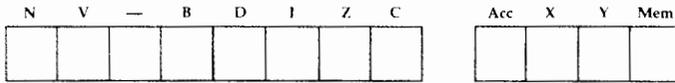
```

ENTRY   LDA   #$00
        STA   PTR
        LDA   #$02
        STA   PTR      ; PTR,PTR+1 = $200
        LDY   #$00
LOOP    LDA   (PTR),Y
        CMP   #$8D      ; CHR = <CR>?
        BEQ   FOUND
        INY
        BNE   LOOP      ; 'TILL Y = $00 AGAIN
DONE    RTS
FOUND   STY   MEM
        BEQ   DONE      ; (ALWAYS)
    
```

JMP: JuMp to Address

DESCRIPTION: Causes program execution to jump to the address specified.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	JMP \$FFff	4C ff FF
Indirect	JMP (\$FFff)	6C ff FF

NOTE: The 6502 has a well documented *bug* regarding the indirect jump. If the jump specified uses pointers which do not cross a page boundary (for example, \$3C0,3C1), then all will go as predicted. If, however, the pointers cross a boundary (such as \$3FE,400), then the assumed bytes *will not be used*. Instead, the address data will be retrieved from locations \$3FF and \$300. That is to say that the high-order byte is not properly incremented within the 6502 so that both bytes are retrieved from the same page of memory. This should be taken into account if such a situation can possibly arise in your routine.

USES: Besides the obvious application of the usual absolute addressed JMP instruction, the indirect JMP is used when creating vectored jumps. The Apples uses many such indirect jumps, the most notable of which are:

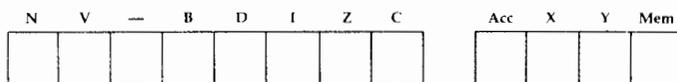
<u>ROUTINE</u>	<u>LOC.</u>	<u>LABEL</u>	<u>VECTOR LOC.</u>	<u>FUNCTION</u>
IRQ + 9:	\$FA49	IRQLOC	\$3FE,3FF	Interrupt Vector
BRK + \$A:	\$FA56	BRKV	\$3F0,3F1	Break Vector
RDKEY + F:	\$FD18	KSWL	\$38,39	Input Vector
COUT:	\$FDED	CSWL	\$36,37	Output Vector

An indirect jump can also be used when writing relocatable code. If the current location of the code can be determined, then an offset can be calculated and the vectors set up so that the JMP will be relative to the current location of the code. See chapter fourteen for more information on these techniques.

JSR: Jump to SubRoutine

DESCRIPTION: The address of the instruction following the JSR is pushed onto the stack. The address following the JSR is then jumped to. When an RTS in the called subroutine is encountered, a return to the location on the stack (the one after the JSR) is done. This is analgous to a GOSUB in Basic.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute Only:	JSR \$FFFF	20 ff FF

USES: JSR is one of the most commonly used instructions, being used to call often needed subroutines. The disadvantage of the instruction is that if the JSRs reference addresses within the code (as opposed to routines external to the program, such as in the Monitor ROM), the code can only be executed at the location for which the code was originally assembled.

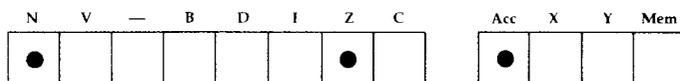
Because the calling address is saved on the stack, a JSR to a known RTS can be done, and the data retrieved to determine where in memory the routine is currently being executed.

See chapter fourteen for more details on both of these topics.

LDA: LoaD Accumulator

DESCRIPTION: Loads the Accumulator with either a specified value, or the contents of the designated memory location. The N (sign) and Z (zero) flags are conditioned when a value with the high bit set is loaded, or when a zero value is loaded.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

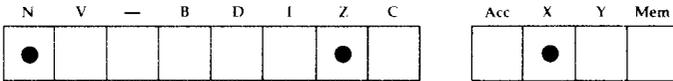
MODE	COMMON SYNTAX	HEX CODING		
Absolute	LDA \$FFFF	AD	ff	FF
Zero Page	LDA \$FF	A5	FF	
Immediate	LDA #\$FF	A9	FF	
Absolute,X	LDA \$FFFF,X	BD	ff	FF
Absolute,Y	LDA \$FFFF,Y	B9	ff	FF
(Indirect),X	LDA (\$FF),X	A1	FF	
(Indirect),Y	LDA (\$FF),Y	B1	FF	
Zero Page,X	LDA \$FF,X	B5	FF	

USES: LDA is probably *the most used* of any instruction. The vast majority of operations center around the Accumulator, and this instruction is used to get data into this important register.

LDX: Load the X-Register

DESCRIPTION: Loads the X-Register with either a specified value, or the contents of the designated memory location. The N (sign) and Z (zero) flags are conditioned when a value with the high bit set is loaded, or when a zero value is loaded.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

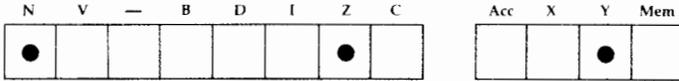
MODE	COMMON SYNTAX	HEX CODING		
Absolute	LDX \$FFFF	AE	ff	FF
Zero Page	LDX \$FF	A6	FF	
Immediate	LDX #\$FF	A2	FF	
Absolute,Y	LDX \$FFFF,Y	BE	ff	FF
Zero Page,Y	LDX \$FF,Y	B6	FF	

USES: This is the primary way in which data is placed into the X-Register. What more can I say?

LDY: LoAD the Y-Register

DESCRIPTION: Loads the Y-Register with either a specified value, or the contents of the designated memory location. The N (sign) and Z (zero) flags are conditioned when a value with the high bit set is loaded, or when a zero value is loaded.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

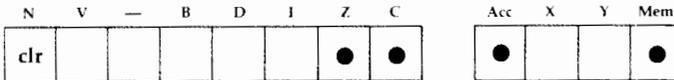
MODE	COMMON SYNTAX	HEX CODING		
Absolute	LDY \$FFFF	AC	ff	FF
Zero Page	LDY \$FF	A4	FF	
Immediate	LDY #\$FF	A0	FF	
Absolute,X	LDY \$FFFF,X	BC	ff	FF
Zero Page,X	LDY \$FF,X	B4	FF	

USES: This is the primary way in which data is placed into the Y-Register. See LDX for additional comments.

LSR: Logical Shift Right

DESCRIPTION: This instruction moves each bit of the Accumulator or memory location specified one position to the right. A zero is forced at the bit 7 position (the high-order bit), and bit 0 falls into the carry. The result is left in the Accumulator or memory location. (See also ROR; also ASL and ROL).

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Accumulator	LSR	4A
Absolute	LSR \$FFff	4E ff FF
Zero Page	LSR \$FF	46 FF
Absolute,X	LSR \$FFff,X	5E ff FF
Zero Page,X	LSR \$FE,X	56 FF

USES: ASL provides an easy way of dividing by a power of two. The corresponding effect in decimal arithmetic is well known: $123/10 = 12.3$ (shift right). As an example:

```

ENTRY   LDA   MEM
        LSR
        LSR           ; DIV BY 2
        STA   MEM           ; DIV BY 2 AGAIN
        LSR           ; MEM = MEM/4
    
```

ASL also provides a fast way of detecting whether a number is odd or even:

```

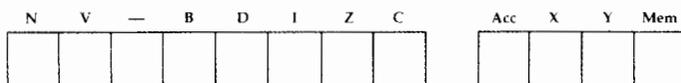
ENTRY   LDA   MEM
        LSR
        BCS   EVEN
        BCC   ODD
    
```

Since bit 0 determines the odd/even nature of a number, this is easily transferred to the carry via the LSR, and then checked via the BCS/BCC instructions.

NOP: No OPeration

DESCRIPTION: Does nothing for one instruction (two cycles). May remind you of some people you know.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING:
Implied Only:	NOP	EA

USES: NOP is used primarily to disable portions of code written by other programmers that you have decided you can live without. A classic example of this is the placing of three NOPs at bytes \$3B,3C, and \$3D on Track n, Sector n, of a standard DOS 3.3 diskette. By the strategic placement of these NOPs, a boot will not force a clear of the language card, thus avoiding the rather monotonous LOADING LANGUAGE CARD message on every boot.

Additionally, NOPs may be used during debugging to disable certain steps, or to create certain timing periods.

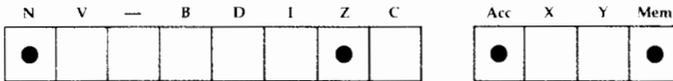
ORA: Inclusive OR with the Accumulator

DESCRIPTION: This instruction takes each bit of the Accumulator and performs a logical OR with each corresponding bit of the specified memory location or immediate value. The result is put back in the Accumulator. The memory location, if specified, is unaffected. Conditions the N (sign) and Z (zero) flags depending on the result. (See ORA and AND also.) Inclusive OR means if *either* or *both* bits are one then the result is one. Only when both bits are zero is the result zero. The truth table used is:

	0	1
0	0	1
1	1	1

Example:
 Accumulator: 0 0 1 1 0 0 1 1
 Memory: 0 1 0 1 0 1 0 1
 Result: 0 1 1 1 0 1 1 1

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	ORA \$FFff	0D ff FF
Zero Page	ORA \$FF	05 FF
Immediate	ORA #\$FF	09 FF
Absolute,X	ORA \$FFff,X	1D ff FF
Absolute,Y	ORA \$FFff,Y	19 ff FF
(Indirect,X)	ORA (\$FF,X)	01 FF
(Indirect,Y)	ORA (\$FF),Y	11 FF
Zero Page,X	ORA \$FF,X	15 FF

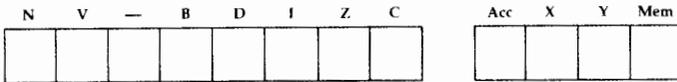
USES: ORA is used primarily as a mask to force ones in specified bit positions. (See AND to force zeros.) To create the mask, a one is put in each bit position which is to be forced. All other positions are set to zero. For example, here is a routine which will set the high bit on any ASCII data going out through COUT:

```
ENTRY LDA DEVICE
      ORA #80 ; %10000000
              ; SET HIGH BIT
      JSR COUT ; $FDED
      RTS
```

PHA: PusH Accumulator

DESCRIPTION: This pushes the contents of the Accumulator onto the stack. The Accumulator and Status Register are unaffected. (See PLA also.)

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	PHA	48

USES: This is one of the most common ways of temporarily storing a byte or two. It is combined with PLA to retrieve the data. Generally speaking, each PHA must be matched by a PLA later in the routine. Otherwise the final RTS of your routine will deliver you, not back to the calling Basic program or immediate mode, but rather off in the weeds, as the saying goes.

Here is an example of a simple store/retrieve operation:

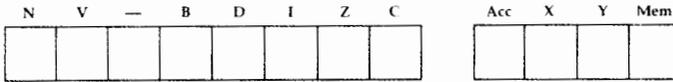
```
ENTRY   LDA   #$80      ; TEST VALUE
        PHA   ; STORE IT
        LDA   #$FF      ; DESTROY ACC.
        PLA   ; RETRIEVE VALUE
        STA   MEM       ; SAVE IT TO LOOK AT
DONE    RTS
```

Another more obscure use of PHA is to set up an artificial JMP by executing an RTS for which a JSR was never done. Providing two PHAs have been done prior to the RTS, the pseudo-jump will be executed. See chapter fourteen for more details on this.

PHP: Push Processor Status

DESCRIPTION: This pushes the Status Register onto the stack for later retrieval. The Status Register itself is unchanged, and none of the registers are affected.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	PHP	08

USES: PHP is done to preserve the Status Register for later testing for a specific condition. This is handy if you don't want to test a flag right then, but the next instruction would ruin what you want to test for. By putting the Status Register on the stack, and then later retrieving it, you can test things like the sign flag or carry when it's most convenient.

ENTRY	CLC		; CLR CARRY
	PHP		; SAVE REG
	SEC		; SET CARRY
	PLP		; RETRIEVE REG
	BCC	DONE	; (ALWAYS!)
	BRK		; (NEVER)
DONE	RTS		

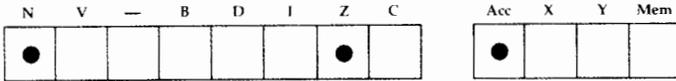
ENTRY	LDA	#\$00	; SET Z FLAG
	PHP		; SAVE REG
	LDA	#\$FF	; DESTROY
	PLP		; RETRIEVE
	BEQ	DONE	; (ALWAYS!)
	BRK		; (NEVER)
DONE	RTS		

Like the PHA instruction, PHP should always be accompanied by an equal number of PLP instructions to keep the Apple happy. Remember: It's not nice to fool the stack!

PLA: Pull Accumulator

DESCRIPTION: This is the converse of the PHA instruction. PLA retrieves one byte from the stack and places it in the Accumulator. This accordingly conditions the N (sign) and Z (zero) flags, just as though a LDA instruction had been done.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	PLA	68

USES: This is combined with PHA to retrieve data from the stack. See PHA for an example of this.

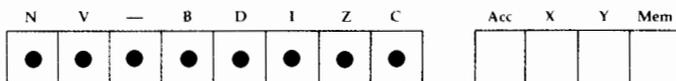
Additionally, PLA can be used to cancel a current RTS, much like a POP in Applesoft Basic. To cancel the most recent RTS, two PLAs are required:

ENTRY	JSR	LEVEL1	
	RTS		; WOULD EXIT HERE NORMALLY
LEVEL1	LDA	#\$00	; ARBITRARY OPERATION
	PLA		
	PLA		; 'POP' RTS
EXIT	RTS		; WILL EXIT ENTIRELY HERE

PLP: Pull Processor Status

DESCRIPTION: This is used after a PHP to retrieve the Status Register data from the stack. The byte is put in the Status Register and all the flags are conditioned corresponding to the status of each bit in the byte placed there. The Accumulator and other registers are unaffected. (See PHP.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	PLP	28

USES: PLP is used much like the retrieve the Status Register *after* a PHP has stored the flags at an earlier time. The examples used for PHP are duplicated here for your convenience:

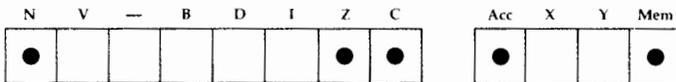
ENTRY	CLC		; CLR CARRY
	PHP		; SAVE REG
	SEC		; SET CARRY
	PLP		; RETRIEVE REG
	BCC	DONE	; (ALWAYS!)
	BRK		; (NEVER)
DONE	RTS		
ENTRY	LDA	#\$00	; SET Z FLAG
	PHP		; SAVE REG
	LDA	#\$FF	; DESTROY
	PLP		; RETRIEVE
	BEQ	DONE	; (ALWAYS!)
	BRK		; (NEVER)
DONE	RTS		

As with the PHA/PLA set, PLPs should always be matched with a corresponding number of PHP instructions, in a one-to-one relationship. Failure to observe this requirement can result in some *very* strange results!

ROL: ROTate Left

DESCRIPTION: This instruction moves each bit of the Accumulator or specified memory location one position to the left. The carry bit is pushed into position 0, and is replaced by bit 7 (the high-order bit). The N (sign) and Z (zero) flags are also conditioned depending on the result of the shift. (See ASL; also ROR and LSR.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Accumulator	ROL	2A
Absolute	ROL \$FFff	2E ff FF
Zero Page	ROL \$FF	26 FF
Absolute,X	ROL \$FFff,X	3E ff FF
Zero Page,X	ROL \$FE,X	36 FF

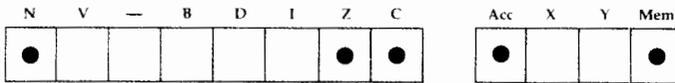
USES: ROL can be used as one of the various methods to test for the high bit set. The disadvantage to testing for the high bit in this way is that the contents must then be restored with a corresponding ROR instruction.

ROL is more often used in combination with ASL in multiply and divide routines.

ROR: ROTate Right

DESCRIPTION: This instruction moves each bit of the Accumulator or memory location specified one position to the right. The carry bit is pushed into position 7 (the high-order bit), and replaced by bit 0. The N (sign) and Z (zero) flags are also conditioned depending on the result of the shift. (See LSR; also ASL and ROL.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Accumulator	ROR	6A
Absolute	ROR \$FFff	6E ff FF
Zero Page	ROR \$FF	66 FF
Absolute,X	ROR \$FFff,X	7E ff FF
Zero Page,X	ROR \$FF,X	76 FF

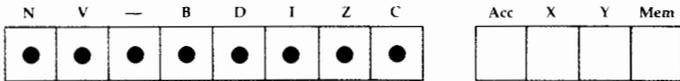
USES: ROR provides an alternate way of testing for the odd/even nature of a number. The carry is tested after the shift to detect whether the number was odd or even.

ROR finds greater use when combined with the shift operations in creating multiply and divide routines.

RTI: ReTurn from Interrupt

DESCRIPTION: This restores both the program counter and the Status Register in preparation to resuming the routine being executed at the time of the interrupt. All flags of the Status Register are reset to the original values.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

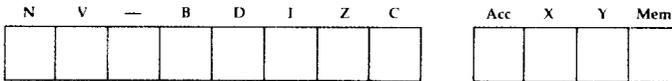
MODE	COMMON SYNTAX	HEX CODING
Implied Only:	RTI	40

USES: RTI is used in much the same way that an RTS would be used in returning from a JSR. After an interrupt has been handled and the background operation performed, the return is done via the RTI command. Usually the user will want to restore the Accumulator, the X-Register and the Y-Register prior to returning.

RTS: ReTurn from Subroutine

DESCRIPTION: This restores the program counter to the address stored on the stack, usually the address of the next instruction after the JSR that called the routine. Analogous to a RETURN to a GOSUB in Basic. (See JSR also.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	RTS	60

USES: RTS is, surprisingly enough, most often used to return from subroutines. It can on occasion be used to simulate a JMP instruction, by using two PHA instructions to put a false return address on the stack, and then executing the RTS. See the section on PHA, and also chapter fourteen for more details.

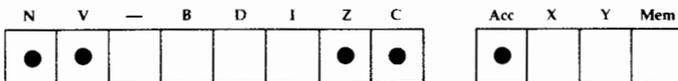
An RTS can be POPed one level by the execution of two PLA instructions.

SBC: SuBtract with Carry

DESCRIPTION: Subtracts the contents of the memory location or a specified value from the Accumulator. The opposite of the carry is also subtracted, and in this instance, the carry is called a borrow. The N (sign), V (overflow), Z (zero), and C (carry) flags are all conditioned by this operation, and are often used to detect the nature of the result of the subtraction. The result of the subtraction is put back in the Accumulator. The memory location, if specified, is unchanged. SBC works for both the binary and BCD arithmetic modes.

IMPORTANT: A SEC should always be done prior to the first SBC operation. This is equivalent to *clearing the borrow*. This is analogous to the CLC done prior to an ADC instruction.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING		
Absolute	SBC \$FFFf	ED	ff	FF
Zero Page	SBC \$FF	E5	FF	
Immediate	SBC #\$FF	E9	FF	
Absolute,X	SBC \$FFFf,X	FD	ff	FF
Absolute,Y	SBC \$FFFf,Y	F9	ff	FF
(Indirect),X	SBC (\$Ff,X)	E1	FF	
(Indirect),Y	SBC (\$Ff),Y	F1	FF	
Zero Page,X	SBC \$Ff,X	F5	FF	

USES: SBC is used most often for either 1) subtracting a constant or memory value from a one-byte memory location, or 2) subtracting a constant or memory value from a two-byte memory location.

One-Byte Subtraction

```
ENTRY    SEC
         LDA    MEM
         SBC    #$80
         STA    MEM
DONE     RTS
```

(MEM = MEM - #\$80)

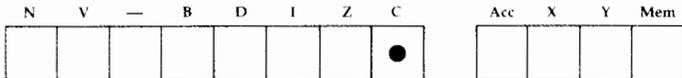
Two-Byte Subtraction

```
ENTRY    SEC
         LDA    MEM
         SBC    #$80
         STA    MEM
         LDA    MEM + 1
         SBC    #$00
         STA    MEM + 1
DONE     RTS
```

(MEM, MEM + 1 = MEM, MEM + 1 - #\$80)

DESCRIPTION: This sets the carry flag of the Status Register.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	SEC	38

USES: SEC is usually used just prior to a SBC operation. The carry is occasionally used though to indicate error (or other) conditions, as is done by RWTS (Read-Write Track Sector) in DOS. In these instances SEC is used to set the carry to indicate an error. This would be detected sometime later in program execution, after a return from RWTS has already been made.

SEC is also sometimes used to force a branch. For example:

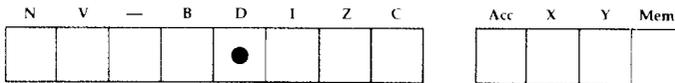
```

SEC
BCS ADDRESS ; (ALWAYS)
    
```

SED: SET Decimal Mode

DESCRIPTION: SED sets the 6502 to the Binary Coded Decimal (BCD) mode, in preparation for a ADC or SBC operation.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	SED	F8

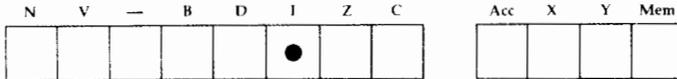
USES: BCD math is used when a greater degree of precision is required. In this mode each four bits of a byte represent one digit of a base ten number. Although not discussed in the articles covered in this volume, here is a brief example of a BCD addition operation:

```
ENTRY  SED                ; SET DEC MODE
        CLC
        LDA  #$25          ; %00101001 = #25
        ADC  #$18          ; %00011000 = #18
        STA  MEM           ; RSLT = %01000011 = #43
        CLD                ; CLR DEC MODE
DONE   RTS
```

SEI: SET Interrupt Disable

DESCRIPTION: SEI is used to disable the interrupt response to an IRQ (a maskable interrupt). This does not disable the response to a NMI (Non-Maskable Interrupt = RESET).

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

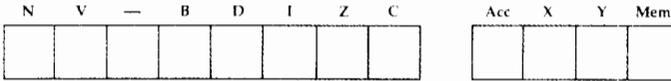
MODE	COMMON SYNTAX	HEX CODING
Implied Only:	SEI	78

USES: SEI is automatically set whenever an interrupt occurs so that no further interrupts can disturb the system while it is going through the \$FFFE,FFFF to \$3FE,3FF vector path. The user is expected to use CLI to re-enable interrupts upon entry to his own interrupt routine. DOS typically does a SEI/CLI operation upon entrance to and exit from RWTS so that interrupts do not interfere with the highly timing-dependent disk read/write routines.

STA: STore Accumulator

DESCRIPTION: Stores the contents of the Accumulator in the specified memory location. The contents of the Accumulator are not changed, nor are any of the Status Register flags.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	STA \$FFff	8D ff FF
Zero Page	STA \$FF	85 FF
Absolute,X	STA \$FFff,X	9D ff FF
Absolute,Y	STA \$FFff,Y	99 ff FF
(Indirect),X	STA (\$FF,X)	81 FF
(Indirect),Y	STA (\$FF,Y)	91 FF
Zero Page,X	STA \$FF,X	95 FF

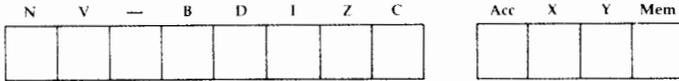
USES: STA is another highly used instruction, being used at the end of many operations to put the final result into a memory location.

In general, the LDA/STA combination is used to transfer bytes from one location to another.

STX: Store the X-Register

DESCRIPTION: STX stores the contents of the X-Register in the specified memory location. The X-Register is unchanged and none of the Status Register flags are affected.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

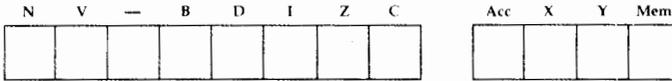
MODE	COMMON SYNTAX	HEX CODING		
Absolute	STX \$FFFF	8E	ff	FF
Zero Page	STX \$FF	86	FF	
Zero Page,Y	STX \$FF,Y	96	FF	

USES: It is occasionally useful to be able to store the contents of the X-Register for later reference. Another fairly common use of STX is in Applesoft's determination of string lengths. After getting data from the input buffer (\$200-2FF) the length of the input string is held in the X-Register, and is saved to a string *descriptor* for later use. See chapter twelve for a listing of a simple INPUT routine.

STY: Store the Y-Register

DESCRIPTION: STY stores the contents of the Y-Register in the specified memory location. The Y-Register is unchanged and none of the Status Register flags are affected.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Absolute	STY \$FFff	8C ff FF
Zero Page	STY \$FF	84 FF
Zero Page,X	STY \$FE,X	94 FF

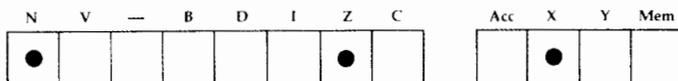
USES: STY is used to store the value of the Y-Register, usually from within string or data scanning loops. For example, here is a routine which returns the position of the first control character in a block of data:

```
ENTRY   LDY   #$00           ; ZERO COUNTER
LOOP    LDA   DATA,Y       ; GET CHARACTER
        BEQ   NOTF          ; CHR=0 = END
        CMP  #$20           ; 'SPC'
        BCS  NXT            ; CHR > CTRL'S
FOUND   STY   POS           ; SAVE Y-REG
DONE    RTS
NXT     INY                   ; Y = Y + 1
        BNE  LOOP          ; 'TILL Y=0 AGAIN.
        BEQ  DONE
NOTF    LDY   #$FF           ; FLAG NOTFOUND
        BNE  FOUND
```

TAX: Transfer Accumulator to X-Register

DESCRIPTION: Puts contents of Accumulator into the X-Register. Does not affect the Accumulator.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	TAX	AA

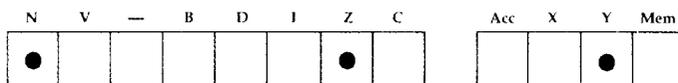
USES: Most simply, TAX is used for transferring data in the manner in which it implies. Equally important however, is its combination with TYA to form a transfer from Y-Register to X-Register function:

ENTRY	LDY	#\$00	;	LOAD Y
	TYA		;	PUT IN A
	TAX		;	PUT IN X

TAY: Transfer Accumulator to Y-Register

DESCRIPTION: Puts contents of Accumulator into the Y-Register. Does not affect the Accumulator.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
------	---------------	------------

Implied Only:	TAY	A8
---------------	-----	----

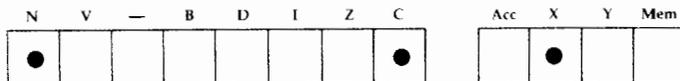
USES: Most often, TAY is used for transferring data from the Accumulator to the Y-Register. Equally important, however, is its combination with TXA to form a transfer from X-Register to Y-Register function:

ENTRY	LDX	#\$00	;	LOAD X
	TXA		;	PUT IN A
	TAY		;	PUT IN Y

TSX: Transfer Stack to X-Register

DESCRIPTION: This puts the contents of the stack pointer into the X-Register. The N (sign) and Z (zero) flags are conditioned. The stack pointer is unchanged.

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	TSX	BA

USES: The most obvious use of TSX is in preserving the value of the stack at a certain point. This could be used in place of the PLAS mentioned under the PLA and RTS headings in attempting to duplicate the equivalent of Basic's POP command, that is to say, a direct return to a different level than the one which had actually called a subroutine. For example:

```

ENTRY   LDA   #$00      ; DUMMY OPERATION
        TSX   ; SAVE CURRENT RETURN PTR.
        JSR   LEVEL1
        RTS   ; NORMAL EXIT, BUT IT
              ; WILL NEVER BE CALLED HERE.
LEVEL1  TXS   ; PUT PTR TO 1ST RETURN BACK.
DONE    RTS   ; EXIT TO MAIN CALLING PROGR.
    
```

Note that this is somewhat dangerous in that you must be very certain as to the actual contents of the stack, and in the knowledge that the data has not been changed by intermediate PHAS and PLAS for instance. Remember that S is only a *pointer* to the stack and does not preserve the return address as such, only its position in the stack.

Another use for TSX is in retrieving data from the stack without having to do a PLA instruction. Although a PLA/PHA/TAX sequence would be transparent to the stack, and accomplish the same results, TSX can be used to retrieve information that is

officially lost at that point. What I am alluding to is retrieving data that is lower in memory than the current stack pointer, and which would be overwritten by the next PHA instruction. One of the prime examples of this is in using a JSR to a known RTS in the Monitor for no other purpose than to be able to immediately retrieve the otherwise transparent return address. This is done so that relocatable code has a way of finding out where it's currently located. See chapter fourteen for a thorough explanation of the technique. For quick reference, here's the basic routine:

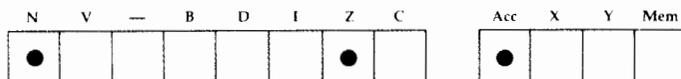
```
ENTRY    JSR    RETURN    ; $FF58
         TSX
         LDA    STACK,X   ; $100,X
         STA    PTR+1
         DEX
         LDA    STACK,X   ; $100,X+1
         STA    PTR       ; PTR,PTR+1 = ENTRY+2.
DONE     RTS
```

CAUTION: Most Step and Trace utilities will not properly trace code like this because of the somewhat *illegal* use of the stack. Strictly speaking, good programming principles dictate that once data is officially off the stack, it is counted as being effectively lost. This is especially true in the case of interrupts, where an interrupt in the middle of the dummy JSR, RTS and retrieval process could produce a completely invalid result in PTR,PTR+1. *Caveat emptor!*

TXA: Transfer X to Accumulator

DESCRIPTION: This puts the contents of the X-Register into the Accumulator, and thus conditions the Status Register just as if a LDA instruction had been executed. The X-Register is unaffected by the operation. (See also TAX.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	TXA	8A

USES: TXA provides a way of retrieving the value in the X-Register for appropriate processing by the program. In the case of string related routines, this is often the length of the string just entered or scanned. The Accumulator can then go about the things it does so well in terms of putting the value into the most useful part of memory. Notice that there are more addressing modes available to STA command, not to mention the overall powers granted the Accumulator in terms of logical operators.

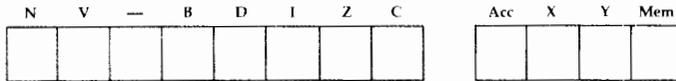
As discussed under TAY, TXA can be combined with TAY to form a TXY-like (transfer X to Y) function like so:

ENTRY	LDX	MEM	;	GET DATA
	TXA		;	PUT IN X
	TAY		;	MOVE TO Y

TXS: Transfer X to Stack

DESCRIPTION: This puts the contents of the X-Register into the stack pointer. None of the Status Register flags are affected, nor is the X-Register itself changed.

FLAGS & REGISTERS AFFECTED: (NONE)



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	TXS	9A

USES: TXS is used to put data directly onto the stack pointer. Because there is no TAS (Accumulator to stack) or even TYS (Y-Register to stack), this is the only way to get a specific byte into the stack pointer. This is usually used in conjunction with TSX to restore previously saved data. In the case of the Applesoft stack fix program, it is used to avoid problems that would otherwise occur if a RESUME were not used after an error had occurred within a FOR-NEXT loop, or GOSUB:

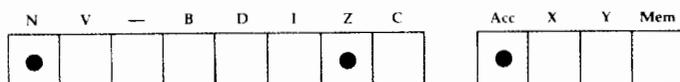
ENTRY	PLA			; GET LOW BYTE OF CURRENT
				; RETURN ADDR.
	TAY			; SAVE INTO Y
	PLA			; GET HIGH BYTE OF
				; RETURN ADDR.
	LDX	ERRSTK		; SDF = S PTR BEFORE ERROR
	TXS			; PUT BEFORE-ERR PTR BACK
	PHA			; PUT HIGH BYTE BACK
	TYA			; GET LOW BYTE IN ACC.
	PHA			; PUT LOW BYTE BACK.
DONE	RTS			; RETURN TO APPLESOFT
				; WITH STACK FIXED.

See also TSX for other applications of TXS.

TYA: Transfer Y to Accumulator

DESCRIPTION: This puts the contents of the Y-Register into the Accumulator, and thus conditions the Status Register just as if a LDA instruction had been executed. The Y-Register is unaffected by the operation. (See also TAX.)

FLAGS & REGISTERS AFFECTED:



ADDRESSING MODES AVAILABLE:

MODE	COMMON SYNTAX	HEX CODING
Implied Only:	TYA	98

USES: TYA provides a way of retrieving the value in the Y-Register for appropriate processing by the program. This comes in handy when scanning a data block, and information regarding certain locations is to be processed. As mentioned under TXA, the Accumulator has far greater flexibility than the Y-Register in terms of addressing modes and logical operators available.

TYA is also combined with TAX to form the equivalent of a TYX (transfer Y to X). The operation has the form of:

ENTRY	LDY	MEM	;	GET DATA
	TYA		;	PUT IN Y
	TAX		;	MOVE TO X



APPENDIX C

6502 MICROPROCESSOR INSTRUCTIONS

ADC	Add memory to Accumulator with Carry	LDA	Load Accumulator with Memory
AND	"AND" Memory with Accumulator	LDX	Load Index X with Memory
ASL	Shift Left One Bit (Memory or Accumulator)	LDY	Load Index Y with Memory
BCC	Branch on Carry Clear	LSR	Shift Right One Bit (Memory or Accumulator)
BCS	Branch on Carry Set	NOP	No Operation
BEQ	Branch on Result Zero	ORA	"OR" Memory with Accumulator
BIT	Test Bits in Memory with Accumulator	PHA	Push Accumulator on Stack
BMI	Branch on Result Minus	PHP	Push Processor Status on Stack
BNE	Branch on Result not Zero	PLA	Pull Accumulator from Stack
BPL	Branch on Result Plus	PLP	Pull Processor Status from Stack
BRK	Force Break	ROL	Rotate One Bit Left (Memory or Accumulator)
BVC	Branch on Overflow Clear	ROR	Rotate One Bit Right (Memory or Accumulator)
BVS	Branch on Overflow Set	RTI	Return from Interrupt
CLC	Clear Carry Flag	RTS	Return from Subroutine
CLD	Clear Decimal Mode	SBC	Subtract Memory from Accumulator with Borrow
CLI	Clear Interrupt Disable Bit	SEC	Set Carry Flag
CLV	Clear Overflow Flag	SED	Set Decimal Mode
CMP	Compare Memory and Accumulator	SEI	Set Interrupt Disable Status
CPX	Compare Memory and Index X	STA	Store Accumulator in Memory
CPY	Compare Memory and Index Y	STX	Store Index X in Memory
DEC	Decrement Memory by One	STY	Store Index Y in Memory
DEX	Decrement Index X by One	TAX	Transfer Accumulator to Index X
DEY	Decrement Memory Y by One	TAY	Transfer Accumulator to Index Y
EOR	"Exclusive-On" Memory with Accumulator	TSX	Transfer Stack Pointer to Index X
INC	Increment Memory by One	TXA	Transfer Index X to Accumulator
INX	Increment Index X by One	TXS	Transfer Index X to Stack Pointer
INY	Increment Index Y by One	TYA	Transfer Index Y to Accumulator
JMP	Jump to New Location		
JSR	Jump to New Location Saving Return Address		

*Appendices C, D, and E reprinted from the APPLE II REFERENCE MANUAL courtesy Apple Computer, Inc.

THE FOLLOWING NOTATION APPLIES TO THIS SUMMARY:

A Accumulator	∇ Logical Exclusive Or
X, Y Index Register	↑ Transfer From Stack
M Memory	↓ Transfer To Stack
C Borrow	→ Transfer To
P Processor Status Register	← Transfer To
S Stack Pointer	∨ Logical OR
✓ Change	PC Program Counter
— No Change	PCH Program Counter High
+ Add	PCL Program Counter Low
∧ Logical AND	OPER Operand
- Subtract	# Immediate Addressing Mode

FIGURE 1. ASL-SHIFT LEFT ONE BIT OPERATION



FIGURE 2. ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

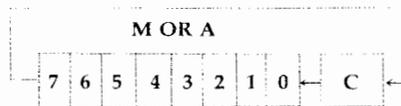
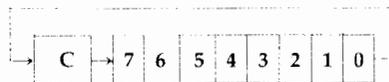


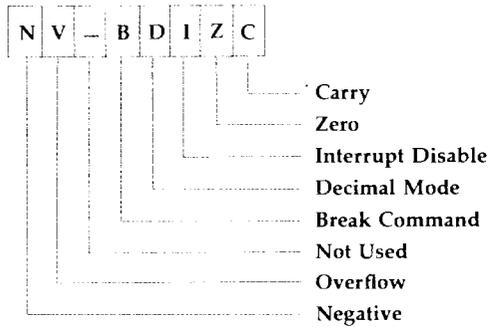
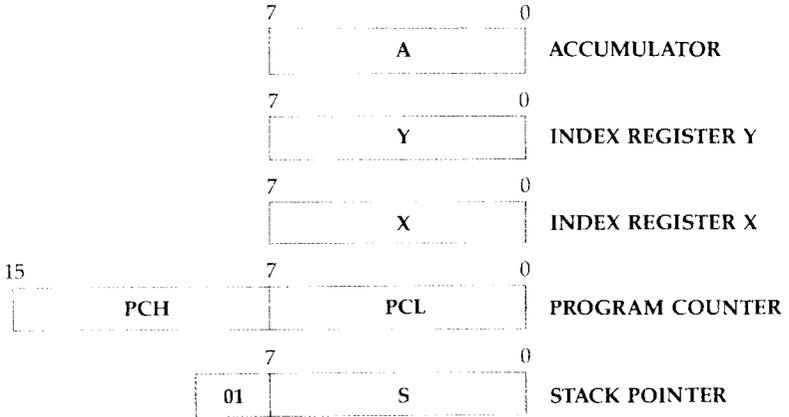
FIGURE 3.



NOTE 1: BIT — TEST BITS

Bits 6 and 7 are transferred to the Status Register. If the result of $A \wedge M$ is zero then $Z=1$, otherwise $Z=0$.

PROGRAMMING MODEL



INSTRUCTION CODES

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX Op Code	No. Bytes	"P" Status Reg. N C Z I D V
ADC Add memory to Accumulator with carry	A-M-C → A.C	Immediate	ADC #Oper	69	2	✓ ✓ ✓ ---
		Zero Page	ADC Oper	65	2	✓ ✓ ---
		Zero Page, X	ADC Oper, X	75	2	✓ ---
		Absolute	ADC Oper	60	3	✓ ✓ ✓ ---
		Absolute, X	ADC Oper, X	70	3	✓ ✓ ---
		Absolute, Y	ADC Oper, Y	79	3	✓ ---
		(Indirect, X)	ADC (Oper, X)	61	2	✓ ---
		(Indirect), Y	ADC (Oper), Y	71	2	✓ ---
AND "AND" memory with Accumulator	A ∧ M → A	Immediate	AND #Oper	29	2	✓ ✓ ---
		Zero Page	AND Oper	25	2	✓ ---
		Zero Page, X	AND Oper, X	35	2	✓ ---
		Absolute	AND Oper	20	3	✓ ✓ ✓ ---
		Absolute, X	AND Oper, X	30	3	✓ ✓ ---
		Absolute, Y	AND Oper, Y	39	3	✓ ---
		(Indirect, X)	AND (Oper, X)	21	2	✓ ---
		(Indirect), Y	AND (Oper), Y	31	2	✓ ---
ASL Shift left one bit (Memory or Accumulator)	(See Figure 1)	Accumulator	ASL A	0A	1	✓ ✓ ✓ ---
		Zero Page	ASL Oper	06	2	✓ ---
		Zero Page, X	ASL Oper, X	16	2	✓ ---
		Absolute	ASL Oper	0E	3	✓ ✓ ✓ ---
		Absolute, X	ASL Oper, X	1E	3	✓ ✓ ---

BCC Branch on carry clear	Branch on C = 0	Relative	BCC Oper	90	2	-----
BCS Branch on carry set	Branch on C = 1	Relative	BCS Oper	80	2	-----
BEQ Branch on result zero	Branch on Z = 1	Relative	BEQ Oper	F0	2	-----
BIT Test bits in memory with Accumulator	A \wedge M, M ₇ \rightarrow N. M _n \rightarrow V	Zero Page Absolute	BIT* Oper BIT* Oper	24 2C	2 3	M ₇ --- M _n
BMI Branch on result minus	Branch on N = 1	Relative	BMI Oper	30	2	-----
BNE Branch on result not zero	Branch on Z = 0	Relative	BNE Oper	D0	2	-----
BPL Branch on result plus	Branch on N = 0	Relative	BPL Oper	10	2	-----
BRK Force Break	Forced Interrupt PC + 2 \downarrow P \downarrow	Implied	BRK*	00	1	--- 1 ---
BVC Branch on overflow clear	Branch on V = 0	Relative	BVC Oper	50	2	-----
BVS Branch on overflow set	Branch on V = 1	Relative	BVS Oper	70	2	-----

Note 1. Bits 6 and 7 are transferred to the Status Register. If the result of A \wedge M is zero, then Z = 1 otherwise Z = 0.

Note 2. A BRK command cannot be masked by setting I.

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX Op Code	No. Bytes	"P" Status Reg. N Z C I D V
CLC Clear carry flag	0 → C	Implied	CLC	18	1	---0---
CLD Clear decimal mode	0 → D	Implied	CLD	D8	1	-0-----
CLI	0 → I	Implied	CLI	58	1	---0---
CLV Clear overflow flag	0 → V	Implied	CLV	B8	1	0-----
CMP Compare memory and Accumulator	A --- M	Immediate	CMP #Oper	C9	2	--- ---
		Zero Page	CMP Oper	C5	2	--- ---
		Zero Page,X	CMP Oper,X	D5	2	--- ---
		Absolute	CMP Oper	CD	3	--- ---
		Absolute,X	CMP Oper,X	DD	3	--- ---
		Absolute,Y	CMP Oper,Y	D9	3	--- ---
		(Indirect,X)	CMP (Oper,X)	C1	2	--- ---
(Indirect),Y	CMP (Oper),Y	D1	2	--- ---		
CPX Compare memory and Index X	X --- M	Immediate	CPX #Oper	E0	2	--- ---
		Zero Page	CPX Oper	E4	2	--- ---
		Absolute	CPX Oper	EC	3	--- ---

CPY Compare memory and Index Y	Y — M	Immediate Zero Page Absolute	CPY #Oper CPY Oper CPY Oper	C0 C4 CC	2 2 3	✓✓-----
DEC Decrement memory by one	M — 1 → M	Zero Page Zero Page,X Absolute Absolute,X	DEC Oper DEC Oper,X DEC Oper DEC Oper,X	C6 D6 CE DE	2 2 3 3	✓✓-----
DEX Decrement Index X by one	X — 1 → X	Implied	DEX	CA	1	✓✓-----
DEY Decrement Index Y by one	Y — 1 → Y	Implied	DEY	88	1	✓✓-----
EOR "Exclusive-Or" memory with Accumulator	A ∨ M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	EOR #Oper EOR Oper EOR Oper,X EOR Oper EOR Oper,X EOR Oper,Y EOR (Oper,X) EOR (Oper),Y	49 45 55 4D 5D 59 41 51	2 2 2 3 3 3 2 2	✓✓-----
INC Increment memory by one	M + 1 → M	Zero Page Zero Page,X Absolute Absolute,X	INC Oper INC Oper,X INC Oper INC Oper,X	E6 F6 EE FE	2 2 3 3	✓✓-----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX Op Code	No. Bytes	"P" Status Reg. N Z C I D V
INX Increment Index X by one	$X + 1 \rightarrow X$	Implied	INX	E8	1	✓ - - - -
INY Increment Index Y by one	$Y + 1 \rightarrow Y$	Implied	INY	C8	1	✓ - - - -
JMP Jump to new location	$(PC + 1) \rightarrow PCL$ $(PC + 2) \rightarrow PCH$	Absolute	JMP Oper	4C	3	- - - - -
		Indirect	JMP (Oper)	6C	3	- - - - -
JSR Jump to new location saving return address	$PC + 2 \downarrow$ $(PC + 1) \rightarrow PCL$ $(PC + 2) \rightarrow PCH$	Absolute	JSR Oper	20	3	- - - - -
		Zero Page	LDA #Oper	A9	2	✓ - - - -
		Zero Page,X	LDA Oper	A5	2	- - - - -
LDA Load Accumulator with memory	$M \rightarrow A$	Absolute	LDA Oper,X	B5	2	- - - - -
		Absolute,X	LDA Oper	AD	3	- - - - -
		Absolute,Y	LDA Oper,X	BD	3	- - - - -
		(Indirect,X)	LDA Oper,Y	B9	3	- - - - -
		(Indirect),Y	LDA (Oper,X)	A1	2	- - - - -
			LDA (Oper),Y	B1	2	- - - - -

LDX						
Load Index X with memory	M → X	Immediate	LDX #Oper	A2	2	✓✓-----
		Zero Page	LDX Oper	A6	2	
		Zero Page, Y	LDX Oper, Y	B6	2	
		Absolute	LDX Oper	AE	3	
		Absolute, Y	LDX Oper, Y	BE	3	
LDY						
Load Index Y with memory	M → Y	Immediate	LDY #Oper	A0	2	✓✓-----
		Zero Page	LDY Oper	A4	2	
		Zero Page, X	LDY Oper, X	B4	2	
		Absolute	LDY Oper	AC	3	
		Absolute, X	LDY Oper, X	BC	3	
LSR						
Shift right one bit (memory or Accumulator)	(See Figure 1)	Accumulator	LSR A	4A	1	0✓✓-----
		Zero Page	LSR Oper	46	2	
		Zero Page, X	LSR Oper, X	56	2	
		Absolute	LSR Oper	4E	3	
		Absolute, X	LSR Oper, X	5E	3	
NOP						
No operation		Implied	NOP	EA	1	-----
ORA						
"OR" memory with Accumulator	A V M → A	Immediate	ORA #Oper	09	2	✓✓-----
		Zero Page	ORA Oper	05	2	
		Zero Page, X	ORA Oper, X	15	2	
		Absolute	ORA Oper	0D	3	
		Absolute, X	ORA Oper, X	1D	3	
		Absolute, Y	ORA Oper, Y	19	3	
		(Indirect, X)	ORA (Oper, X)	01	2	
		(Indirect), Y	ORA (Oper), Y	11	2	

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX Op Code	No. Bytes	"P" Status Reg. N Z C I D V
PHA Push Accumulator on stack	A ↓	Implied	PHA	48	1	-----
PHP Push processor status on stack	P ↓	Implied	PHP	08	1	-----
PLA Pull Accumulator from stack	A ↑	Implied	PLA	68	1	-----
PLP Pull processor status from stack	P ↑	Implied	PLP	28	1	From Stack
ROL Rotate one bit left (memory or Accumulator)	(See Figure 2)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROL A ROL Oper ROL Oper,X ROL Oper ROL Oper,X	2A 26 36 2E 3E	1 2 2 3 3	-----
ROR Rotate one bit right (memory or Accumulator)	(See Figure 3)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	ROR A ROR Oper ROR Oper,X ROR Oper ROR Oper,X	6A 66 76 6E 7E	1 2 2 3 3	-----
RTI Return from Interrupt	P ↑ PC ↑	Implied	RTI	40	1	From Stack

RTS Return from subroutine	PC ↑, PC + 1 → PC	Implied	RTS	60	1	-----
SBC Subtract memory from Accumulator with borrow	A - M - \bar{C} → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	SBC #Oper SBC Oper SBC Oper,X SBC Oper SBC Oper,X SBC Oper,Y SBC (Oper,X) SBC (Oper),Y	E9 E5 F5 ED FD F9 E1 F1	2 2 2 3 3 3 2 2	✓✓✓---✓
SEC Set carry flag	1 → C	Implied	SEC	38	1	--1----
SED Set decimal mode	1 → D	Implied	SED	F8	1	-----1-
SEI Set interrupt disable status	1 → I	Implied	SEI	78	1	---1---
STA Store Accumulator in memory	A → M	Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	STA Oper STA Oper,X STA Oper STA Oper,X STA Oper,Y STA (Oper,X) STA (Oper),Y	85 95 8D 9D 99 81 91	2 2 3 3 3 2 2	-----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX Op Code	No. Bytes	"P" Status Reg. N Z C I D V
STX Store Index X in memory	X → M	Zero Page Zero Page, Y Absolute	STX Oper STX Oper, Y STX Oper	86 96 8E	2 2 3	-----
STY Store Index Y in memory	Y → M	Zero Page Zero Page, X Absolute	STY Oper STY Oper, X STY Oper	84 94 8C	2 2 3	-----
TAX Transfer Accumulator to Index X	A → X	Implied	TAX	AA	1	✓✓-----
TAY Transfer Accumulator to Index Y	A → Y	Implied	TAY	A8	1	✓✓-----
TSX Transfer stack pointer to Index X	S → X	Implied	TSX	BA	1	✓✓-----
TXA Transfer Index X to Accumulator	X → A	Implied	TXA	8A	1	✓✓-----
TXS Transfer Index X to stack pointer	X → S	Implied	TXS	9A	1	-----
TYA Transfer Index Y to Accumulator	Y → A	Implied	TYA	98	1	✓✓-----

HEX OPERATION CODES

00	-- BRK	2F	-- NOP
01	-- ORA -- (Indirect, X)	30	-- BMI
02	-- NOP	31	-- AND -- (Indirect), Y
03	-- NOP	32	-- NOP
04	-- NOP	33	-- NOP
05	-- ORA -- Zero Page	34	-- NOP
06	-- ASL -- Zero Page	35	-- AND -- Zero Page, X
07	-- NOP	36	-- ROL -- Zero Page, X
08	-- PHP	37	-- NOP
09	-- ORA -- Immediate	38	-- SEC
0A	-- ASL -- Accumulator	39	-- AND -- Absolute, Y
0B	-- NOP	3A	-- NOP
0C	-- NOP	3B	-- NOP
0D	-- ORA -- Absolute	3C	-- NOP
0E	-- ASL -- Absolute	3D	-- AND -- Absolute, X
0F	-- NOP	3E	-- ROL -- Absolute, X
10	-- BPL	3F	-- NOP
11	-- ORA -- (Indirect), Y	40	-- RTI
12	-- NOP	41	-- EOR -- (Indirect, X)
13	-- NOP	42	-- NOP
14	-- NOP	43	-- NOP
15	-- ORA -- Zero Page, X	44	-- NOP
16	-- ASL -- Zero Page, X	45	-- EOR -- Zero Page
17	-- NOP	46	-- LSR -- Zero Page
18	-- CLC	47	-- NOP
19	-- ORA -- Absolute, Y	48	-- PHA
1A	-- NOP	49	-- EOR -- Immediate
1B	-- NOP	4A	-- LSR -- Accumulator
1C	-- NOP	4B	-- NOP
1D	-- ORA -- Absolute, X	4C	-- JMP -- Absolute
1E	-- ASL -- Absolute, X	4D	-- EOR -- Absolute
1F	-- NOP	4E	-- LSR -- Absolute
20	-- JSR	4F	-- NOP
21	-- AND -- (Indirect, X)	50	-- BVC
22	-- NOP	51	-- EOR -- (Indirect), Y
23	-- NOP	52	-- NOP
24	-- BIT -- Zero Page	53	-- NOP
25	-- AND -- Zero Page	54	-- NOP
26	-- ROL -- Zero Page	55	-- EOR -- Zero Page, X
27	-- NOP	56	-- LSR -- Zero Page, X
28	-- PLP	57	-- NOP
29	-- AND -- Immediate	58	-- CLI
2A	-- ROL -- Accumulator	59	-- EOR -- Absolute, Y
2B	-- NOP	5A	-- NOP
2C	-- BIT -- Absolute	5B	-- NOP
2D	-- AND -- Absolute	5C	-- NOP
2E	-- ROL -- Absolute	5D	-- EOR -- Absolute, X

5E	-- LSR -- Absolute, X	90	-- BCC
5F	-- NOP	91	-- STA -- (Indirect), Y
60	-- RTS	92	-- NOP
61	-- ADC -- (Indirect, X)	93	-- NOP
62	-- NOP	94	-- STY -- Zero Page, X
63	-- NOP	95	-- STA -- Zero Page, X
64	-- NOP	96	-- STX -- Zero Page, Y
65	-- ADC -- Zero Page	97	-- NOP
66	-- ROR -- Zero Page	98	-- TYA
67	-- NOP	99	-- STA -- Absolute, Y
68	-- PLA	9A	-- TXS
69	-- ADC -- Immediate	9B	-- NOP
6A	-- ROR -- Accumulator	9C	-- NOP
6B	-- NOP	9D	-- STA -- Absolute, X
6C	-- JMP -- Indirect	9E	-- NOP
6D	-- ADC -- Absolute	9F	-- NOP
6E	-- ROR -- Absolute	A0	-- LDY -- Immediate
6F	-- NOP	A1	-- LDA -- (Indirect, X)
70	-- BVS	A2	-- LDX -- Immediate
71	-- ADC -- (Indirect), Y	A3	-- NOP
72	-- NOP	A4	-- LDY -- Zero Page
73	-- NOP	A5	-- LDA -- Zero Page
74	-- NOP	A6	-- LDX -- Zero Page
75	-- ADC -- Zero Page, X	A7	-- NOP
76	-- ROR -- Zero Page, X	A8	-- TAY
77	-- NOP	A9	-- LDA -- Immediate
78	-- SEI	AA	-- TAX
79	-- ADC -- Absolute, Y	AB	-- NOP
7A	-- NOP	AC	-- LDY -- Absolute
7B	-- NOP	AD	-- Absolute
7C	-- NOP	AE	-- LDX -- Absolute
7D	-- ADC -- Absolute, X	AF	-- NOP
7E	-- ROR -- Absolute, X	B0	-- BCS
7F	-- NOP	B1	-- LDA -- (Indirect), Y
80	-- NOP	B2	-- NOP
81	-- STA -- (Indirect, X)	B3	-- NOP
82	-- NOP	B4	-- LDY -- Zero Page, X
83	-- NOP	B5	-- LDA -- Zero Page, X
84	-- STY -- Zero Page	B6	-- LDX -- Zero Page, Y
85	-- STA -- Zero Page	B7	-- NOP
86	-- STX -- Zero Page	B8	-- CLV
87	-- NOP	B9	-- LDA -- Absolute, Y
88	-- DEY	BA	-- TSX
89	-- NOP	BB	-- NOP
8A	-- TXA	BC	-- LDY -- Absolute, X
8B	-- NOP	BD	-- LDA -- Absolute, X
8C	-- STY -- Absolute	BE	-- LDX -- Absolute, Y
8D	-- STA -- Absolute	BF	-- NOP
8E	-- STX -- Absolute	C0	-- CPY -- Immediate
8F	-- NOP	C1	-- CMP -- (Indirect, X)

C2	--- NOP	E1	--- SBC --- (Indirect), X
C3	--- NOP	E2	--- NOP
C4	--- CPY --- Zero Page	E3	--- NOP
C5	--- CMP --- Zero Page	E4	--- CPX --- Zero Page
C6	--- DEC --- Zero Page	E5	--- SBC --- Zero Page
C7	--- NOP	E6	--- INC --- Zero Page
C8	--- INY	E7	--- NOP
C9	--- CMP --- Immediate	E8	--- INX
CA	--- DEX	E9	--- SBC --- Immediate
CB	--- NOP	EA	--- NOP
CC	--- CPY --- Absolute	EB	--- NOP
CD	--- CMP --- Absolute	EC	--- CPX --- Absolute
CE	--- DEC --- Absolute	ED	--- SBC --- Absolute
CF	--- NOP	EE	--- INC --- Absolute
D0	--- BNE	EF	--- NOP
D1	--- CMP --- (Indirect), Y	F0	--- BEQ
D2	--- NOP	F1	--- SBC --- (Indirect), Y
D3	--- NOP	F2	--- NOP
D4	--- NOP	F3	--- NOP
D5	--- CMP --- Zero Page, X	F4	--- NOP
D6	--- DEC --- Zero Page, X	F5	--- SBC --- Zero Page, X
D7	--- NOP	F6	--- INC --- Zero Page, X
D8	--- CLD	F7	--- NOP
D9	--- CMP --- Absolute, Y	F8	--- SED
DA	--- NOP	F9	--- SBC --- Absolute, Y
DB	--- NOP	FA	--- NOP
DC	--- NOP	FB	--- NOP
DD	--- CMP --- Absolute, X	FC	--- NOP
DE	--- DEC --- Absolute, X	FD	--- SBC --- Absolute, X
DF	--- NOP	FE	--- INC --- Absolute, X
E0	--- CPX --- Immediate	FF	--- NOP



APPENDIX D

SOME USEFUL MONITOR SUBROUTINES

Here is a list of some useful subroutines in the Apple's Monitor and Autostart ROMs. To use these subroutines from machine language programs, load the proper memory locations or 6502 registers as required by the subroutine and execute a JSR to the subroutine's starting address. It will perform the function and return with the 6502's registers set as described.

\$FDED **COUT** **Output a character**

COUT is the standard character output subroutine. The character to be output should be in the Accumulator. COUT calls the current character output subroutine whose address is stored in CSW (locations \$36 and \$37), usually COUT1 (see below).

\$FDF0 **COU1** **Output to screen**

COU1 displays the character in the Accumulator on the Apple's screen at the current output cursor position and advances the output cursor. It handles the control characters RETURN, linefeed, and bell. It returns with all registers intact.

\$FE80 **SETINV** **Set Inverse mode**

Sets Inverse video mode for COU1. All output characters will be displayed as black dots on a white background. The Y-Register is set to \$3E; all others are unchanged.

\$FE84 **SETNORM** **Set Normal Mode**

Sets Normal video mode for COU1. All output characters will be displayed as white dots on a black background. The Y-Register is set to \$FE; all others are unchanged.

\$FD8E **CROUT** **Generate a RETURN**

CROUT sends a RETURN character to the current output device.

\$FD8B **CROUT1** **RETURN with clear**

CROUT1 clears the screen from the current cursor position to the edge of the text window, then calls CROUT.

\$FDDA **PRBYTE** **Print a hexadecimal byte**

This subroutine outputs the contents of the Accumulator in hexadecimal on the current output device. The contents of the Accumulator are scrambled.

\$FDE3 **PRHEX** **Print a hexadecimal digit**

This subroutine outputs the lower nibble of the Accumulator as a single hexadecimal digit. The contents of the Accumulator are scrambled.

\$F941 **PRNTAX** **Print A and X in hexadecimal**

This outputs the contents of the Accumulator and X-Register as a four-digit hexadecimal value. The Accumulator contains the first byte output, the X-Register contains the second. The contents of the Accumulator are usually scrambled.

\$F948 **PRBLNK** **Print 3 spaces**

Outputs three blank spaces to the standard output device. Upon exit, the Accumulator usually contains \$A0, the X-Register contains zero.

\$F94A **PRBL2** **Print many blank spaces**

This subroutine outputs from 1 to 256 blanks to the standard output device. Upon entry, the X-Register should contain the number of blanks to be output. If X = \$00, then PRBL2 will output 256 blanks.

\$FF3A **BELL** **Output a "bell" character**

This subroutine sends a bell (CTRL G) character to the current output device. It leaves the Accumulator holding \$87.

\$FBDD **BELL1** **Beep the Apple's speaker**

This subroutine beeps the Apple's speaker for .1 second at 1KHz. It scrambles the Accumulator and X-Register.

\$FD0C **RDKEY** **Get an input character**

This is the standard character input subroutine. It places a flashing input cursor on the screen at the position of the output cursor and jumps to the current input subroutine whose address is stored in KSW (locations \$38 and \$39), usually KEYIN (see below).

\$FD35 **RDCHAR** **Get an input character or ESC code**

RDCHAR is an alternate input subroutine which gets characters from the standard input, but also interprets the eleven escape codes.

\$FD1B **KEYIN** **Read the Apple's keyboard**

This is the keyboard input subroutine. It reads the Apple's keyboard, waits for a keypress, and randomizes the random number seed. When it gets a keypress, it removes the flashing cursor and returns with the keycode in its Accumulator.

\$FD6A **GETLN** **Get an input line with prompt**

GETLN is the subroutine which gathers input lines. Your programs can call GETLN with the proper prompt character in location \$33; GETLN will return with the input line in the input buffer (beginning at location \$200) and the X-Register holding the length of the input line.

\$FD67 GETLNZ Get an input line

GETLNZ is an alternate entry point for GETLN which issues a carriage return to the standard output before falling into GETLN (see above).

\$FD6F GETLN1 Get an input line, no prompt

GETLN1 is an alternate entry point for GETLN which does not issue a prompt before it gathers the input line. If, however, the user cancels the input line, either with too many backspaces or with a CTRL X, then GETLN1 will issue the contents of location \$33 as a prompt when it gets another line.

\$FCA8 WAIT Delay

This subroutine delays for a specific amount of time, then returns to the program which called it. The amount of delay is specified by the contents of the Accumulator. With A the contents of the Accumulator, the delay is $\frac{1}{4}(26 + 27A + 5A^2)$ μ seconds. WAIT returns with the Accumulator zeroed and the X and Y-Registers undisturbed.

\$F864 SETCOL Set Low-Res Graphics color

This subroutine sets the color used for plotting on the Low-Res screen to the color passed in the Accumulator.

\$F85F NEXTCOL Increment color by 3

This adds 3 to the current color used for Low-Res Graphics.

\$F800 PLOT Plot a block on the Low-Res Screen

This subroutine plots a single block on the Low-Res screen of the prespecified color. The block's vertical position is passed in the Accumulator, its horizontal position in the Y-Register. PLOT returns with the Accumulator scrambled, but X and Y unmolested.

\$F819 HLINE Draw a horizontal line of blocks

This subroutine draws a horizontal line of blocks of the predetermined color on the Low-Res screen. You should call HLINE with the vertical coordinate of the line in the Accumulator, the leftmost horizontal coordinate in the Y-Register, and the rightmost horizontal coordinate in location \$2C. HLINE returns with the Accumulator and Y scrambled, X intact.

\$F828 VLINE Draw a vertical line of blocks

This subroutine draws a vertical line of blocks of the predetermined color on the Low-Res screen. You should call VLINE with the horizontal coordinate of the line in the Y-Register, the top vertical coordinate in the Accumulator, and the bottom vertical coordinate in location \$2D. VLINE will return with the Accumulator scrambled.

\$F832 CLRSCR Clear the entire Low-Res screen

CLRSCR clears the entire Low-resolution Graphics screen. If you call CLRSCR

while the video display is in Text mode, it will fill the screen with inverse-mode "@" characters. CLRSCR destroys the contents of the Accumulator and Y.

\$F836 CLRTOP Clear the top of the Low-Res Screen

CLRTOP is the same as CLRSCR (above), except that it clears only the top 40 rows of the screen.

\$F871 SCRN Read the Low-Res screen

This subroutine returns the color of a single block on the Low-Res screen. Call it as you would call PLOT (above). The color of the block will be returned in the Accumulator. No other registers are changed.

\$FB1E PREAD Read a Game Controller

PREAD will return a number which represents the position of a game controller. You should pass the number of the game controller (0 to 3) in the X-Register. If this number is not valid, strange things may happen. PREAD returns with a number from \$00 to \$FF in the Y-Register. The Accumulator is scrambled.

\$FF2D PRERR Print "ERR"

Sends the word "ERR", followed by a bell character, to the standard output device. The Accumulator is scrambled.

\$FF4A IOSAVE Save all registers

The contents of the 6502's internal registers are saved in locations \$45 through \$49 in the order A-X-Y-P-S. The contents of the Accumulator and X are changed; the decimal mode is cleared.

\$FF3F IOREST Restore all registers

The contents of the 6502's registers are loaded from locations \$45 through \$49.

APPENDIX E

KEYS AND THEIR ASSOCIATED ASCII CODES

Key	Alone	CTRL	SHIFT	Both
space	\$A0	\$A0	\$A0	\$A0
0	\$B0	\$B0	\$B0	\$B0
1!	\$B1	\$B1	\$A1	\$A1
2"	\$B2	\$B2	\$A2	\$A2
3#	\$B3	\$B3	\$A3	\$A3
4\$	\$B4	\$B4	\$A4	\$A4
5%	\$B5	\$B5	\$A5	\$A5
6&	\$B6	\$B6	\$A6	\$A6
7'	\$B7	\$B7	\$A7	\$A7
8(\$B8	\$B8	\$A8	\$A8
9)	\$B9	\$B9	\$A9	\$A9
:*	\$BA	\$BA	\$AA	\$AA
;+	\$BB	\$BB	\$AB	\$AB
,<	\$AC	\$AC	\$BC	\$BC
- =	\$AD	\$AD	\$BD	\$BD
.>	\$AE	\$AE	\$BE	\$BE
/?	\$AF	\$AF	\$BF	\$BF
A	\$C1	\$81	\$C1	\$81
B	\$C2	\$82	\$C2	\$82
C	\$C3	\$83	\$C3	\$83
D	\$C4	\$84	\$C4	\$84
E	\$C5	\$85	\$C5	\$85
F	\$C6	\$86	\$C6	\$86
RETURN	\$8D	\$8D	\$8D	\$8D
G	\$C7	\$87	\$C7	\$87
H	\$C8	\$88	\$C8	\$88
I	\$C9	\$89	\$C9	\$89
J	\$CA	\$8A	\$CA	\$8A
K	\$CB	\$8B	\$CB	\$8B
L	\$CC	\$8C	\$CC	\$8C
M	\$CD	\$8D	\$DD	\$9D
N	\$CE	\$8E	\$DE	\$9E
O	\$CF	\$8F	\$CF	\$8F
P@	\$D0	\$90	\$C0	\$80
Q	\$D1	\$91	\$D1	\$91
R	\$D2	\$92	\$D2	\$92
S	\$D3	\$93	\$D3	\$93

All codes are given in hexadecimal.

<u>Key</u>	<u>Alone</u>	<u>CTRL</u>	<u>SHIFT</u>	<u>Both</u>
T	\$D4	\$94	\$D4	\$94
U	\$D5	\$95	\$D5	\$95
V	\$D6	\$96	\$D6	\$96
W	\$D7	\$97	\$D7	\$97
X	\$D8	\$98	\$D8	\$98
Y	\$D9	\$99	\$D9	\$99
Z	\$DA	\$9A	\$DA	\$9A
→	\$88	\$88	\$88	\$88
←	\$95	\$95	\$95	\$95
ESC	\$9B	\$9B	\$9B	\$9B

ASCII SCREEN CHARACTERS

Decimal	Inverse				Flashing				(Control)				Normal				(Lowercase)	
	Hex	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	
	\$00	\$10	\$20	\$30	\$40	\$50	\$60	\$70	\$80	\$90	\$A0	\$B0	\$C0	\$D0	\$E0	\$F0		
0	\$0	@	P		0	@	P		0	@	P		0	@	P		0	
1	\$1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	!	1	
2	\$2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	"	2	
3	\$3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	#	3	
4	\$4	D	T	\$	4	D	T	\$	4	D	T	\$	4	D	T	\$	4	
5	\$5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	%	5	
6	\$6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	&	6	
7	\$7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	'	7	
8	\$8	H	X	(8	H	X	(8	H	X	(8	H	X	(8	
9	\$9	I	Y)	9	I	Y)	9	I	Y)	9	I	Y)	9	
10	\$A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	*	:	
11	\$B	K	[+	;	K	[+	;	K	[+	;	K	[+	;	
12	\$C	L	\	/	<	L	\	/	<	L	\	/	<	L	\	/	<	
13	\$D	M		_	=	M		_	=	M		_	=	M		_	=	
14	\$E	N	^	.	>	N	^	.	>	N	^	.	>	N	^	.	>	
15	\$F	O	_	/	?	O	_	/	?	O	_	/	?	O	_	/	?	

INDEX

absolute addressing, 19, 51, 52
Accumulator, 6-7, 18-19, 41, 42
ADC, 76-79, 148
addition, 76-83
address, 4, 9, 10, 13, 41
addressing, modes, 19, 51-54
address pointers, 53, 82
AND, 107-111
Apple DOS Manual, 94
Apple II Plus, 6
Applesoft, 6, 13, 15, 17, 35, 39, 53,
59, 62, 126-128, 147
Applesoft II Basic Reference Manual, 95
Apple, structure of, 1-12
Apple II Reference Manual, 6, 14,
16-17, 46, 49, 53, 110
ASCII, 44, 100
ASCII Screen Character Set, 20
ASL, 104-105
assemblers, 2, 7, 13-22
assembly language programs, 7
asterisks, 18

base sixteen numbers, 4-7, 24
base two numbers, 24-25, 75-76
BCC, 43
BCS, 43
bell modification, 102
BEQ, 31-32
binary numbers, 24-25, 75-76
bit, 76, 103
BIT, 111-112, 148
BNE, 27
booting, 90
borrow, 83
branch instructions, 27-28, 31-34, 43
branch offsets, 33
break message, 3, 6
buffer pointer, 98
buffers, 90, 98, 139
bytes, 4, 10, 23, 25, 75-76, 79-80, 92

carry (C) flag, 42-43, 77, 82, 83, 99
CATALOG, 90

catalog keypress modifications,
100-102
CH, 135
clear borrow, 83
CMP, 42
command field, 18
comment field, 18
compare command, 42
complements, 85-86
counters, 23, 26-29
COUT, 34-37, 44, 49
CV, 135

data storage, 54-59
decrementing, 26-27
delays, 62-69
delimiters, 17
directory, 93
disassembly, 8-9
disk access, 89-102, 121-128
disk controller cards, 149
diskette organization, 91-99
Disk Operating System (DOS), 17,
35, 89-102
disk, reading/writing files to, 129-143
disk-volume modification, 99-100
dollar (\$) sign, 4
DOS commands, 122-124
DOS error codes, 99
DOS memory organization, 90
DOS modifications, 99-102
DOS Tool Kit, 55
drive error, 99
dummy return address, 151
duration, 65-67

editor/assemblers, 13
editors, 17
EOR, 112-114
EQU, 17, 18
ERR byte, 99

error codes, 99
 exclusive OR, 112–114

fields, 17–18
 file buffer, 98
 flags, 23, 25, 26–27, 41, 42–43, 77,
 82, 83, 86–88, 99, 111–112, 148
 forced branch statement, 148

game paddles, 37–39, 68–69
 GETLN, 125

hard-sectoring, 92
 hex, 4–7, 24
 HEX, 55
 hexadecimal notation, 4–7, 24
 high-order byte, 10, 80–81
 HOME, 20–21, 42

immediate addressing, 19, 51, 52
 implicit/implied addressing, 51, 52
 inclusive OR, 112–113
 incrementing, 26–27
 indexed addressing, 51, 52–53
 indexed indirect addressing, 51, 54
 indexing holes, 92–93
 indirect indexed addressing, 51, 53
 indirect jumps, 160–161
 input buffer, 98
 input routines, 124–128
 Integer Basic Apple, 6, 13, 15, 17, 35,
 62
 INVFLG, 110
 IOB table, 94–98
 I/O routines, 45–49, 121–128

JMP commands, 147–149, 160–161
 JSR, 9–10, 146, 150–152, 155, 156
 JSR/RTS dummy return address, 152,
 156
 JSR simulations, 154–158

keyboard, sound generation from,
 67–68
 keyboard buffers, 47

labels, 17
 LDA, 19
 line numbers, 9, 17
 load/store opcodes, 18–19
 Logical Operator Demo Program,
 116–119
 logical operators, 106–111
 loops, 23, 26–29, 31–32
 low-order byte, 10, 80–81
 LSR, 104, 105

machine code, 2
 machine language, 7
 mask, 111
 math operations, 75–87
 MAXFILES, 90
 memory chips, 1
 memory locations, 1–5
 memory map, 3
 memory range, 5, 8
 Mini-Assembler, 13–15
 mnemonics, 2, 9, 13
 Monitor, 6, 7–11
 Monitor programs for I/O routines,
 45–46

negative numbers, 84–88
 NO BUFFERS AVAILABLE error, 90
 nonrelocatable code, 145–147
 NOP, 62–63

number complements, 85–86
 numbers. *See* binary numbers,
 hexadecimal notation, math
 operations, negative numbers,
 positive numbers
 number (#) sign, 19, 52

OBJ, 18

object code, 16–17
 offset branch, 33
 ones' complement, 85
 opcodes, 2, 9, 17, 18–19
 operands, 9, 18
 ORA, 112–113
 ORG, 18
 overflow (V) flag, 111, 148–149

paddles, 37–39, 68–69
 page, 5
 parity, 105
 PHA, 73–74, 156, 158
 pitch, 62, 64–67
 PLA, 73–74, 156
 positive numbers, 84–88
 print routines, 121–124
 pseudo opcodes, 18, 55

ramp tone pattern, 164
 read error, 99
 reading/writing files on disk, 129–143
 Read Only Memory, 6
 registers, 6–7, 18–19, 23, 25, 41–42,
 53–54, 76–77, 86–87
 relative addressing, 51, 52
 relocatable code, 145–147
 reverse branches, 33
 ROL, 105–106
 ROM, 6
 ROR, 105–106
 RTS, 10, 17, 151–152, 156
 RWTS, 91, 93–99

SBC, 83
 SEC, 83
 sector interleaving, 92
 sectors, 92–98
 self-modifying code, 158–160
 set borrow, 83
 seven bit code, 44
 shift operators, 103–106
 sign bit, 84–86
 sign (N) flag, 86–88
 6502 microprocessor, 1–2, 6–7, 16, 41
 skew factor, 92
 SLOT, 105
 slot/drive values, 102
 soft-sectoring, 92
 sound generation routines, 61–69,
 164–169
 source code, 16–17
 source listings, 16–17
 speaker, 61–62
 STA, 19, 49
 stack, 71–74, 124, 150–154
 stack pointer, 71, 151–152
 Status Register, 23, 25, 41, 76–77,
 86–87
 stepping techniques, 149
 strobe, 46
 subtraction, 83–84
 syntax, 17

TAY, 51, 52
 text files, 142–143
 Text Screen Map, 20
 time delays, 62–69
 tone, 63–64, 68
 tracks, 92–98
 transfer commands, 39–40
 two-byte addition, 80–83
 twos' complement, 86

UCMD, 98
 USLOT, 98, 105

vector, 95, 109
 volume mismatch error, 99
 volume number (VOL), 98
 volume table of contents (VTOC), 93
 VTAB, 135

wrap-around, 26–27, 36, 77
 write-protect label, 143

XFER, 127

X-Register, 6-7, 18-19, 41, 53-54,
106

Y-Register, 6-7, 18-19, 41, 53-54, 106

zero (Z) flag, 25, 26-27, 41, 111-112

zero page addressing, 51, 52, 73



ROGER WAGNER

This volume includes the first fifteen installments of "Everyone's Guide to Assembly Language," by Roger Wagner, originally published in *Softalk* magazine. Expanded, corrected, indexed, and with an introduction by the author, *Assembly Lines: The Book* is better than the original.

Creator of several popular programs for the Apple computer—*Apple Doc*, the *Correspondent* and *Roger's Easel*—Roger Wagner is the president of Southwestern Data Systems based in Santee, California. In addition to Wagner's programs, SDS also markets such successful programs for the Apple as *ASCII Express*, *SpeedStar*, *Merlin*, and *Z-Term*.

Wagner is a graduate of San Diego State University with a degree in physical science, and he has taught math and science at Mountain Empire High School in Southern California.

Concern for the end user permeates Wagner's dealing with the microcomputer world. His philosophy calls for utility programs that equal industry standards and at the same time educate the user.

An accomplished programmer and teacher, Wagner brings this philosophy to fruition within the pages of this book.

In clear, concise language, Roger Wagner makes the basics of assembly language palatable to the experienced and novice programmer alike.

As thousands of Apple owners and *Softalk* readers already know, for learning assembly language one can do no better than Southwestern Data Systems' young president, Roger Wagner.