The Visible Computer: 6502

Software Masters™

3330 Hillcroft/Suite BB
Houston, Texas 77057

For a period of ninety days after purchase, Software Masters will replace defective Visible Computer: 6502 program disks free of charge. Replacement cost after ninety days is $5.00. No other warranty is expressed or implied.

Software Masters is a division of C & C Software Designs, Inc.

The Visible Computer: 6502 was written by Charles Anderson.

Introduction

The Visible Computer: 6502 Machine Language Teaching System combines this manual with a 6502 simulator program to provide a systematic way to learn machine language programming on Apple II computers.

The Visible Computer is a program that teaches programming. The title is a takeoff on those transparent plastic models of men that once (and maybe still do) populated sixth grade classrooms. Like The Visible Man, The Visible Computer lets you see into a place not normally accessible to the eye. Places like chest cavities and accumulators, address latches and pancrei. Unlike The Visible Man, TVC requires no assembly, no careful painting, and no smelly airplane glue.

PREREQUISITES FOR THE USER OF TVC

This manual assumes some familiarity (not to be confused with expertise) with Basic. Programming is programming, and the more experience you have with any form of it the better. It presupposes no prior exposure to machine language, and includes preliminary chapters on binary and hexadecimal numbering systems and computer operations.

HARDWARE REQUIREMENTS

To run The Visible Computer, you will need a 48K Apple II with either Applesoft in ROM (Apple II Plus), or a 16K RAM card. A printer is optional.

SCOPE

Many of the dozen or so books that profess to teach 6502 machine language work so hard at touching all the bases, from floating point arithmetic to control programs for hypothetical daisy wheel printers, that they skimp on the fundamental job of **delivering the concepts.** The Visible Computer is designed to get you over the initial hurdles of machine language programming, not to present algorithms for controlling elevator systems.

Learning everything there is to learn in this manual will not qualify you to immediately go to work at Microsoft writing 6502 Cobol compilers. But if you apply yourself, it will get you to the point where you will be able to develop independently in your area of interest, be

it arcade games, chess programs, or new and wonderful operating systems. And who knows, someday the Microsoft recruiter might just give you a call.

HOW THE MANUAL IS ORGANIZED

Chapters 1, 2, and 3 are the standard introductory fare of Hex, Binary, and Computer Block Diagrams. They may be skipped by those who have already been through eleven discussions of hex and binary (and if they see one more block diagram of a computer, they'll scream).

The TVC program disk is not booted until Chapter 4. It wouldn't be a bad idea to skip there quickly right now and make sure that your TVC disk can boot–but go no farther.

The heart of the course is Chapters 6 through 14, where you'll work through a series of progressively more difficult 6502 machine language programs contained on the TVC disk. By the end of Chapter 14 you will have read about, and seen demonstrated, nearly all of the 56 6502 instructions, and will have earned the honorary title of **TVC Master.**

Chapter 15 puts it all together in three programs that do the kinds of things people learn machine language to do–sorting, high resolution graphics, and tone generation. The concept of assembly language is presented.

Lastly, Chapter 16 tries to wean you from the handholding of previous chapters. There's a suggested reading list, a quick rundown on the options available in assemblers, and pointers on interfacing machine language routines with Basic.

Table of Contents

1.

What is Machine Language?

And If It's So hard, Why Do People Use It? This is a fair question, and if you haven't asked it yet you probably should have. Before we get into the **how**s of machine language we're going to touch on the **why**s.

Of all the programming languages used on the Apple II, from Fortran to Pascal to Applesoft, the language closest by far to an Apple's silicon heart is 6502 machine language. Although later chapters will present a more formal definition, for now it suffices to say that 6502 machine language is the fundamental language of Apple computers. Not a moment passes during an Apple's powered-on lifetime when it is **not** executing 6502 machine language programs. In fact, languages like Basic and Pascal are nothing but clever ruses to save poor humans from the wicked binary ways of 6502 processors.

As to the widespread rumor that machine language programming is more difficult than programming in Basic, consider these two sets of instructions for building a cedar fence in your backyard.

Basic

Using 6' by 6" cedar slats, with supporting posts every 8 feet, build a fence enclosing your back yard.

Machine Language

Drive to lumber yard. Purchase 722 6' by 6" cedar slats. Load into truck. Drive home. Unload truck. Start at northeast corner of back yard. Dig a hole three feet deep. Get post from pile. Insert post into hole. Cement post. Move 8 feet west. If not yet at corner, dig a hole three feet deep. Get post from pile. Insert post into hole. . .

Is the second set of instructions more difficult than the first? Not really. It **looks** more involved, and certainly took longer to write down, but the individual jobs that make up the second paragraph are simplicity itself. "Move 8 feet west", "Get post from pile". So it is with machine language. Working from a limited palette of about 50 simple instructions, we achieve complex results by combining them cleverly.

Machine language programmers have to take smaller steps to get where they're going. That means it takes longer. As a rule of thumb, 10 times as long as working in Basic. Economically speaking, it costs 10 times as much to hire a programmer to get Job X accomplished in machine language as it does getting Job X done in Basic. Furthermore, almost anything you can do in machine language can be done in Basic.

So why do people knock themselves out learning and writing machine language programs? Two main reasons: 1. **For speed.** 2. **For more speed.** Machine language programs execute 10-100 times faster than similar programs written in Basic. (Purists and other curmudgeons will object to this statement, and there **is** something to be said for the fact that unless someone had written the machine language program named Applesoft, Basic would not exist, even as an alternative.) Is speed that important? It depends.

In an accounting program, where the computer spends most of its time waiting for the operator to hit a key, or the printer to finish, or a disk drive to get something, blinding speed is not important. We hear phrases like "printer bound" and "floppy bound". A program that is printer bound can only be speeded up by buying a faster printer. Writing accounting programs in assembly language, then, results in programs that wait for user input at very high speed, and cost 10 times as much to develop as acceptably speedy programs written in Basic. Clearly, an idea whose time has not come.

But sometimes speed is desireable, even critical. In animation, for example. Most of the latest generation of Apple game programs could not function written in Basic. They would do **something**–but things would be so slow as to make a Choplifter sortie last 24 hours, and a single revolution of the blades a minute. So game programs, especially the arcade type, are one place where we need the speed of machine language.

Many times the best tact is a combination of Basic and machine language. Take the accounting application from a minute ago. Most of it can be written in slow-to-execute, but fast-to-program Basic. Certain time consuming jobs will be allocated to machine language. Jobs like sorting.

Sorting programs written in Basic, for those of you who have avoided learning about such things thus far in your programming careers (and your time is coming), are slow. **Really** slow. Sorting a list of 1,000 employee numbers into a stack with the biggest at the bottom and the smallest at the top takes at least two minutes, and maybe as many as 10, depending on what method we tackle the problem with. (The methods available range from the crude–read easy–to the complex. Graduate students as yet unborn will earn their degrees with programs that sort .01% more efficiently than some other program.)

Two minutes is an important length of time to an operator of an accounting package, and ten minutes is an eternity. The strategy followed by the smart programmer, then, is to use Basic for everything except the sort itself–and pass that job to a hard-to-write, but breathtakingly fast machine language program. After 10 seconds (or one or two, depending on how fancy a method we use), the Basic program is handed on a silver platter a sorted list of employee numbers.

Sharing the work between machine language and Basic is a good technique, employed by countless Apple programs, including TVC itself. Mostly Basic, machine language where you need the speed.

To sum up: The best reason for programming an Apple II in machine language is to speed up a process that would be too slow otherwise. Conversely, except as a learning exercise, it is a waste of time to use machine language for something that would be acceptably fast written in Basic.

2.

Alternate Numbering Systems

If you bought The Visible Computer with the hope that it would somehow save you the effort of climbing Mount Hexadecimal, picking you up magically and dropping you safely into the valley of machine language programming on the other side, sorry, no can do.

People don't use binary and hexadecimal numbers to make machine language programming easy; they use them to make it **feasible.** Although it is arguable, barely, that one could learn some machine language without ever learning hex, a person who went that route would find himself working three times as hard for one third as much as the guy who learned the tools of the trade first and the programming second.

If you are fuzzy on the hex and binary numbering systems, **do not skip this chapter.** Learning machine language is a cumulative process and skipping a critical part of the foundation is a good way to build an unstable building.

A TWELVE IS A 12 IS A 1100

Most 20th century Americans (i.e., you and me) agree that the symbols "1" and "2", printed together, like this:

12

have a certain numeric meaning. Specifically, "12" represents the quantity of dots printed here:

. .. .

Or this many commas:

, , , , ,, , , , , ,,

But there is nothing intrinsically "12-like" about these symbols sitting next to each other. If we wanted to form a club that said from now on, "*" would stand for 12 and "#" for 17, we could. Without fear of arrest. Let's do that. You and I will be the charter members of the "* = 12 and # = 17" Club.

Until further notice, "*" represents this many things:

,, ,, ,,, ,,,, ,

and "#", this many:

, ,, , , , ,,, ,,,,, , , ,

How many eggs in a dozen? Very good, * is correct. What fab group had a 1964 hit called "She was Just #"? Right again, the Beatles. Although we'd have to work fairly hard at it the first couple of months, eventually it would become almost as natural as the old way.

Except when we're doing math. What's * times #? Even for people as smart and good looking as members of the club, getting that answer is pretty tough. Whereas everyone elses' notation, "12 times 17", lends itself to computational tricks like carrying and partial products, our representation gives not a clue to the answer. We'd have to either memorize all the combinations of multiplications and divisions for * and #, or give up comparison shopping forever.

This situation isn't as farfetched as you might imagine. Consider the Roman Empire. For all its accomplishments, Rome's state-of-the-art method for representing numbers was what we now call Roman numerals. (Although I suppose they simply referred to them as 'numbers'). As with our club's method, Roman numerals are okay for some things, (like the names of popes and book report outlines), and lousy for others, like calculations.

It's a wonder they built Bridge I considering how hard their engineers had to work to do this simple division:

XXIX / IV

Stop and think about it. If **you** had to solve this problem you'd probably proceed like this: Convert both parts into "normal" notation. Divide using conventional techniques. Finally, convert the answer back to Roman numerals. Unfortunately, "normal" notation hadn't been invented yet, and wouldn't for another 500 years.

When an Arabian astronomer devised a better system around 500 AD, Roman numerals had had it. Not only was the new Arabic notation better for representing long numbers than the Roman method, it greatly facilitated performing arithmetic. Let's see why the Arabic method is so powerful. Numbers written with this system can be methodically broken into their component parts.

Fourth Digit	Third Digit	Second Digit	First Digit
10^3	10^2	10^1	10^0
1000	100	10	1

The number 3,479 breaks into:

3 X 1000		4 X 100		7 X 10		9 X 1
3000	+	**400**	+	**70**	+	**9**

209 is:

2 X 100		0 X 10		9 X 1
200	+	**0**	+	**9**

The value of a digit depends on its **position** in the number. The value is always ten times the value of the same digit one position to the right, and one-tenth the value of the same digit one position to the left. The biggest problem keeping previous designers of numerical representation schemes from implementing a system like this was that they never saw a need for a character to represent **0**, the quantity nothing. Without zero to serve as a placeholder, you can't have positional representation.

The usefulness of the Arabic positional system has nothing to do with the symbols that form the counting alphabet. **1**'s, **2**'s, and **3**'s aren't any better or worse than **I**'s, **V**'s, and **X**'s. It's the positional concept that makes it better. We will refer to this ingenious, and for most of us, familiar scheme henceforth not as "Arabic Positional", but as decimal. Base 10. Because 10 is the magic number that each position is based on.

But this quantity of things:

* * * * * * * * * *

is by no means magic in the grand scheme of the universe. No more "round" or "even" than this many things:

***** *** ****

So why **do** we use 10 as the magic number of our positional notation? Class? Anyone have a guess? Right. In all probability, because people have 10 fingers, and for millions of years, fingers were all we had for representing numbers. On ET's home planet we can be reasonably sure their positional numbering system is based on the number:

* * * * * * * *

The decimal numbering system has remained just about unchanged for 1,500 years because it is an extremely useful way of representing numbers. There exist computational methods that allow 12-year-olds to calculate 5 digit products and sums and even square roots, with nothing but paper and pencil.

And in all probability it will be popular 1,500 years from now, even though the advent of the $4 calculator makes some of its best features (ease of manual calculation) moot. If Roman numerals could have hung in there until the Age of Cheap Calculators, they would have been in good shape. But there is one area where decimal falls flat on its well-known face. Computers. Especially machine language programming of computers.

Because of the way they work, computers have a working vocabulary of only two digits. It's easy to make an electronic device store a one or a zero, much harder to make one that can store 0 though nine. We can easily build a sensor that can detect whether a light bulb is on or off. Far more complex is a sensor that can consistently detect 10 discrete levels of brightness.

Computers need a two digit, or **binary**, positional numbering system. The two digits are 1 and 0. If computers used lightbulbs as their active storage element, we might use the terms "On" and "Off".

We don't need unique digits to represent 2-9 because they can be formed by combinations of 1's and 0's, just as decimal doesn't need unique digits to represent values greater than 9. **1001** is a perfectly acceptable way to express the same quantity represented in decimal as **9.**

BINARY POSITIONAL CHART

Fourth Digit	Third Digit	Second Digit	First Digit
2^3	2^2	2^1	2^0
8	4	2	1

1010 breaks into:

1 * 8 + 0 * 4 + 1 * 2 + 0 * 1 = 10 decimal

1110 is

1 * 8 + 1 * 4 + 1 * 2 + 0 * 1 = 14 decimal

Binary numbers can be added and subtracted with the same techniques we know for decimal.

```
                 1           11    <= Carrys
   1010        1010        1001
 + 0100      + 0010    +   0011
   ----        ----        ----
   1110        0100        1100
```

Carrys happen a lot in binary addition. And borrows are common in subtraction. Otherwise, nothing too taxing about binary arithmetic.

Here's a formula (the only one in this book) to calculate the largest number **X** you can store in **n** positions of base **B** numbers:

$$X = B^n - 1 .$$

Representing even modest quantities in binary tends to be wasteful of paper. Counting to 10:

0
01
10
11
100
101
110
111
1000
1001
1010

Four digits of binary don't hold values as large as four positions of decimal. In fact, it's not even close; 15 vs. 9,999. To handle the range of numbers we encounter in day to day life takes a lot of binary digits.

531 = 1000010011. 1,119 = 10001011111.

Numbers like this have a tendancy to confuse people. It helps a little if we group clumps of four digits into "nibbles". Four binary digits, or **bits**, make a nibble. Eight bits make a byte (isn't that **precious**!)

In nibble form, 531 is **0010 0001 0011.**

Not good, but better. Is there a better way? An intermediate step between binary-loving computers and decimal-trained, 10 fingered, tree-loving human beings?

SUPPOSE PHONES HAD TWO BUTTONS

Suppose the phone company decided to release a new, improved telephone. "**DigiPhone, The Phone of Tomorrow**", with only two buttons, 1 and 0. They'd have a big advertising campaign to convince people that it would be faster, more modern, better in every way than the old phones.

Everybody's telephone number is converted to binary: 844-7171 becomes 1000-0000 1110 0100 1100 0011. Area codes get expanded from three digits to 10, enough to cover all 1,000 possible area codes. The phone book doubles in size, but that's no problem—they make the type twice as small.

But they've misread the American people, who don't like the new system. Not at all. They say it's almost impossible to correctly dial, much less memorize, a phone number like:

(0010 1100 1001) 0101-0000 1000 1001 0011 1000

A one digit mistake and you're calling a McDonald's in Kansas City instead of your grandmother in Rockford, Illinois.

The phone company has already built 286 million DigiPhones and they're not about to junk them. But they do offer a compromise. They take out full page ads in newspapers across the country:

> " **Here's what we'll do, America.** We'll go back to our old phone books and publish everyone's number in the old 10 button form. Numbers will be easy to remember, just like before. When you get ready to call someone, convert it to 2 button format and make the call."
>
> " **Converting your old fashioned decimal telephone number into modern, digital form is a breeze.** First, try to divide 8,388,608 into your phone number. If it fits, the first digit is one, if it doesn't, the first digit is zero. Next, divide 4,194,304 into the remainder. If it fits, the second digit is a one. Otherwise, it's a zero. Next, . . ."

People let the phone company know that a ten minute calculator session everytime they needed to make a call was a less than perfect solution. A company think tank huddled for a week, and a second compromise announced.

A new phone book, with numbers listed in a new, fairly easy to remember format. A format that also possesses the property of converting easily, almost automatically, into binary. The great breakthrough? Something called **hexadecimal.** Easier to remember for people than binary. Not quite as easy as the decimal they've been using since the first grade, but much easier than binary. And easy to convert into and out of binary for dialing.

Whereas decimal has 10 digits in its counting alphabet, and binary two, hexadecimal has **16**. This is a problem because we don't have symbols laying around to represent these six new digits. Although we could have invented new symbols, it was expedient to use something that most people (and typewriters) already knew how to write. They decided that the first six letters of the alphabet would stand for the missing digits. (Music set a precedent when it stole letters to stand for Do-Re-Mi-Fa, etc.)

Not only does it convert easily, it saves paper; Four digits of hex can represent numbers as large as 65,535 ($16^4 - 1$). We can get by with six digit phone numbers. Phone book type can be larger. Huzzah.

Armed with the idea that sometimes letters can be numbers, examine this chart that counts in all three bases.

Decimal	Binary	Hex	Decimal	Binary	Hex
0	0000 0000	00	16	0001 0000	10
1	0000 0001	01	17	0001 0001	11
2	0000 0010	02	18	0001 0010	12
3	0000 0011	03	19	0001 0100	13
4	0000 0100	04	20	0001 0101	14
5	0000 0101	05	21	0001 0110	15
6	0000 0110	06	22	0001 0111	16
7	0000 0111	07	23	0001 1000	17
8	0000 1000	08	24	0001 1001	18
9	0000 1001	09	25	0001 1010	19
10	0000 1010	0A	26	0001 1011	1A
11	0000 1011	0B	27	0001 1100	1B
12	0000 1100	0C	28	0001 1101	1C
13	0000 1101	0D	29	0001 1110	1D
14	0000 1110	0E	30	0001 1111	1E
15	0000 1111	0F	31	0001 1111	1F
			32	0010 0000	20

See the relationship between hex and binary? One hex digit can stand in for each binary nibble. Once you've memorized the hex equivalent of each nibble, conversion between hex and binary is a snap.

A hex telephone number like $4-56CA0 becomes:

4 = 0100
5 = 0101
6 = 0110
C = 1100
A = 1010
0 = 0000

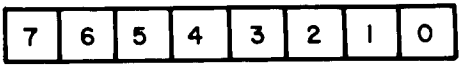
Put it all together and you've got the binary equivalent,

0100-0101 0110 1100 1010 0000

Converting from binary to hex is equally simple. Substitute the hex equivalent of each nibble, and you've got it.

0011 0111 0001 1000 1100 0010
3 7 1 8 C 2 or 3718C2.

In computers, common lengths of binary numbers are 8 and 16 digits. The bits in a byte are numbered from right to left as shown below:



Bit 0 is called the **least significant bit (LSB)**, and bit 7, the **most significant bit (MSB).**

There are two problems left in acclimating ourselves to this new numbering system: First, how do we tell whether a number like 345 is hex or decimal just by looking at it, and second, how on earth do we pronounce something like F3C0?

To clear up the former situation it was agreed by 6502 programmers to always precede hex numbers with a dollar sign ("$"). This convention will be followed throughout this book. It has nothing to do with Applesoft's use of "$" to indicate string variables. 345 is a decimal number. $345 is a hex number equal to 837 in decimal.

For most of you the long term problem will be how to internally verbalize hex numbers containing letters. No one conquers this entirely, but as a rule, call the thing by each character if it contains a "funny" number. "$C13" is "cee-one-three". Also, try calling $F000 "Ef-thousand" and $C00, "Cee-hundred".

THE LOGICAL OPERATORS

Binary numbers have some properties that go beyond just representing decimal values and wasting paper. Numbers as simple as 1 and 0 lend themselves to some special tricks involving what are called the logical (or **boolean**, after George Boole, 19th century English mathematician) operators. These operators are **and**, **or**, and **exclusive or**.

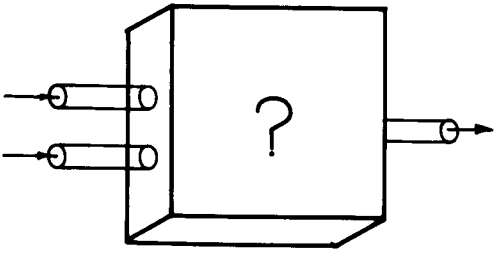
The logical operators are not unlike the four common arithmetic operators, plus, minus, multiply, and divide. The biggest difference is that they operate on binary numbers only one digit long. An example of a logical operation is:

1 AND 1

or,

0 OR 1.

Frequently, logical operations are shown schematically, as a "black box" with two inputs, a mysterious internal process, and one output.



THE RULES

An AND operation yields a 1 if and only if both inputs are 1.

An OR operations yields a 1 if one or both inputs are 1.

AN EOR operation yields a 1 if the inputs are different.

That's it for the rules. Not much to them.

You've probably used logical operators in Basic programs without knowing it. The Basic IF statement is based on logical operations.

IF (expression is logical 1) THEN do this.

IF A > B THEN GOTO 1000

To handle this line, Basic first resolves the assertion portion of the command (A>B) to a simple logical value; either 1 or 0. if A is less than or equal to B, 0 is inserted. If A is greater than B, a 1 is inserted. By definition, 0's cause THEN statements to be bypassed, and 1's cause them to be executed.

IF A OR B THEN GOTO 1000

will cause a branch to 1000 if either variable A or variable B is non-zero. (Basic considers anything non-zero to be a 1.) It is also possible to say things like:

T = (B > C) * 14

If Basic executes this line when B is greater than C, Variable T will be assigned the value 14, because (B > C) will be replaced with the logical value 1. If B is not greater than C, T will be zero.

You can string logical operators together to form complex statements.

IF A > B or (FLAG and G < 14) THEN GOTO 1000

This comes natural to most people, because we phrase such expressions everyday:

"If I can find it and you give me the money, I'll buy it."

"If it doesn't rain tomorrow or if you leave the car, I'll go downtown"

"If the copy machine is working, or Bill has the flyers printed and I can get them in time, you'll get your brochure. "

FINAL EXAM/ALTERNATE NUMBERING SYSTEMS 3201

Fill in the blanks of this Hex to Binary/Binary to Hex conversion chart **without** referring to this manual.

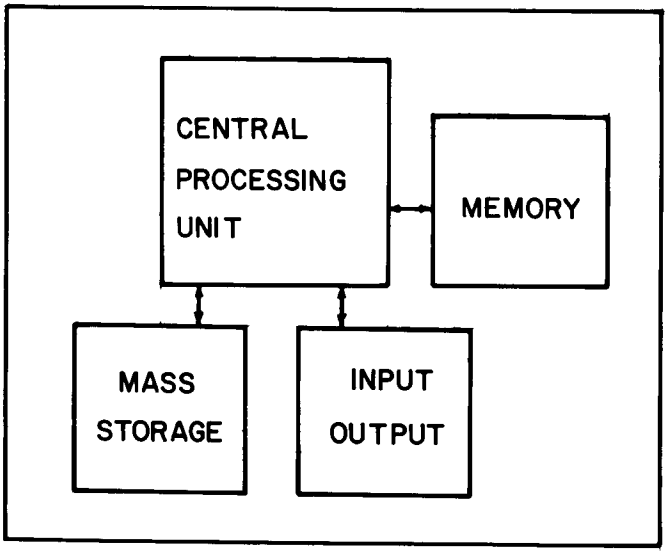
BINARY	HEX	BINARY	HEX
1001 1010	___	_____	$F0
1111 1011	___	_____	$02
0000 0001	___	_____	$CA
1111 0000	___	_____	$0C
1100 1101	___	_____	$DD
0101 1010	___	_____	$11
1011 1011	___	_____	$E6

Perform these logical operations.

0 AND 1 = ___	1 EOR 1 = ___
1 AND 0 = ___	0 EOR 0 = ___
0 OR 1 = ___	1 AND 1 = ___
1 OR 1 = ___	0 OR 0 = ___
1 EOR 1 = ___	1 EOR 0 = ___
0 EOR 1 = ___	0 AND 0 = ___

3.
Hardware

A Control Data Corporation Cyber 6600 computer is big enough to fill a medium size house. An Apple II Plus doesn't weigh 10 pounds soaking wet (perish the thought). But these machines have a lot in common; in fact, at the block diagram level they are identical.



CENTRAL PROCESSING UNIT (CPU)

The absolute monarch of every computer is the CPU. The CPU makes all the decisions and puts the other components through their paces. Although there are almost as many different central processing units as there are computers, they each share the same duties of control, decision, and calculation.

MEMORY

Memory is second fiddle to the CPU, but still an indispensable member of the team. The CPU goes to memory for the stream of numbers that govern its operation, a machine language program. The fundamental operation of a computer is the CPU **reading numbers out of memory and writing numbers into memory.** Were it not for the need to occasionally communicate with human beings, CPU and memory could happily function without the other two components.

MASS STORAGE

Things like disk drives, cassette tape, and punch cards--places where the CPU can store and retrieve numbers, but not in the fast, intimate way it works with memory. Mass storage is for numbers that are not needed immediately and there isn't room for at the moment in memory. Mass storage usually has desirable financial qualities compared to memory; it costs less per byte.

INPUT/OUTPUT (I/O)

These are the links that connect the binary, numerical world of a computer with the world of people. Things like printers, keyboards, and game paddle controllers.

MORE SPECIFICALLY, THE APPLE II

An Apple II uses the 6502 microprocessor as its CPU. A microprocessor is an integrated circuit (IC) that has an entire CPU squeezed onto it. The CPU of a large mainframe computer may consist of more **boards** than an Apple has IC's. The 6502 was introduced in 1975 by a small California company, MOS Technology. MOS Technology was subsequently bought out by Commodore, and the 6502 is now manufactured by them and two second sources, Rockwell and Synertek. If you pry the lid off an Apple and look for the big 40 pin IC mounted horizontally just in front of the expansion slots, that's the 6502. Somewhere in the maze of fuzzy characters written on it you should see the numbers 6502. Probably made by Synertek.

Apple is not the only company around using the 6502 in their machines. Atari video games and home computers, Commodore machines like PET and VIC, the AIM and KIM single board computers and Ohio Scientific systems all use it. A lot of devices that are not full blown computers, like smart video terminals, have a 6502 calling the shots.

Currently, the 6502's greatest challenger for supremacy in the 8 bit microprocessor field is Zilog's Z-80. Although the 6502 and the Z-80 have more similarities than differences, the Z-80 is considered a **register-oriented** processor and the 6502 a **memory-oriented** processor. The Z-80 has more on-board storage, and the 6502 more flair in dealing with memory.

The 6502 is said to be an eight bit microprocessor because it deals with memory in eight bit chunks. This number comes ultimately from the fact that eight of the 40 pins on the 6502 handle the transfer of binary numbers into and out of the 6502. Each leg transmits and receives the electronic equivalent of one and zero.

Eight bits, one byte, is enough to represent numbers from 0 through 255. Although this sounds like a serious limitation, a little programming, teamed with the 6502's tremendous speed, allows the use of numbers as big as we want.

16 of the pins on the 6502 chip are used to specify addresses to memory. This is equivalent to a 16 digit binary number, and means there are 65,536 (2^{16}) memory cells **potentially** addressable by a 6502. Memory can be thought of as a series of numbered cubbyholes, 65,536 of them, maybe in a giant roll-top desk; each cubbyhole has a slip of paper that can hold a number from 0 - 255 (actually, eight tiny slips of paper just big enough for a one or a zero). Reading memory is the act of first specifying the cubbyhole and then reading back the number stored there. Writing to memory involves locating a specific cubbyhole, and scribbling a new number on the slip of paper, erasing whatever was there before.

The 6502 in a powered-up Apple II is continuously engaged in a fast dialog with its memory. If you were to put your ear against the main circuit board, and were very quiet, you might hear something like this (then again, you might not):

6502 Memory–Give me the contents of cell 45601.

Memory Okay. That number is... uh, 123.

6502 Now I need the contents of 45602.

Memory That's going to be. . . 234.

6502 Uh huh. Very interesting. (He thinks for a microsecond or two). Okay, I need you to put 116 in location 1121.

Memory You got it.

There are three basic types of memory cells that make up the memory mechanism a 6502 will find attached to its address and data pins, and not all are "full service".

MEMORY CELL TYPES

RAM is the most useful type of memory cell. RAM stands for Random Access Memory, meaning you can ask one microsecond for location 3 and the next microsecond for location 6,319. As opposed to cassette tape, a **sequential** storage medium. If you want the last byte on a tape you must go through the first 22,000 to get it. A memory cell implemented with RAM will obediently read and write data at the command of the CPU. Although there are ways to build RAM cells that don't, when you turn off the power to an Apple II's RAM circuits, within a few milliseconds, the numbers stored there disappear. (Who among us hasn't gnashed his teeth because of this at least once.)

Enter the second type of memory, ROM. Like RAM, a memory cell implemented with ROM contains numbers that the 6502 can read (in any order; it's just as random access as RAM). The difference is that the numbers in a ROM cell are permanently engraved at the factory, and cannot be changed, no matter how many times the CPU tries to write to it. Thus the acroynm: Read Only Memory. This is both a liability and a blessing. It's not very flexible (what if you want to do something with the computer that doesn't need these numbers?) but it has the endearing quality of withstanding being turned off without losing the numbers stored there.

I/O locations are the third type of Apple memory cell. These are memory locations that are tied to elements of the computer other than the CPU-ROM/RAM clique. I/O locations let the 6502 communicate with the rest of the machine. Some I/O addresses allow external devices to communicate with the 6502: In the rolltop desk analogy, these cubbyholes have a trap door in the back, and some third party is responsible for the numbers that appear there. The 6502 looks at the slip of paper in a cell marked "Keyboard" when it needs to know what key is being pressed. If the 6502 tries to write to this cell, it doesn't work; only the keyboard can change the number stored here.

Other I/O locations are address dependent switches. These cubbyholes have trip wires that trigger a hidden mechanism whenever we try to read or write that cell. Any read or write of cell 49,200 (an I/O address labeled "Speaker") causes a speaker somewhere (in a drawer, I suppose) to make a sound. The simple act of addressing this cell,

regardless of whether with a read or write operation, trips the wire and makes the desired event happen.

DIVIDE AND CONQUER

A useful way to organize 65,536 (**64K**, where 1 **K** = 2^{10} = 1,024) memory locations is to group them into 256 "pages" of 256 locations each. Think of memory as a book with 256 pages, and 256 words (bytes) on each page. Page 3 is locations 768-1024, or $300 - $3FF. The page concept is a natural for hex representation, as every address breaks neatly into a **page** and a **location** within the page. Memory cell $3411 is the $11th byte of page $34.

Just because the 6502 has the **potential** to access 65,536 memory locations doesn't mean that every 6502 in the world can count on having that many locations available to it. The engineer who wants to use a 6502 as the brains of the microwave oven he's designing may decide that he doesn't need more than 1,000 bytes of ROM and 100 bytes of RAM to build the world's smartest microwave oven. The 6502 that finds itself installed in such a microwave still has the **capacity** to access 65,536 locations, but only a thousand are really there. If it tries to access one of the unimplemented addresses, it's like a robot in a Datsun factory blindly trying to arc-weld a 280-Z stalled 10 feet up the assembly line. It **thinks** it's reading an instruction at location $C000, but it's seeing random, arbitrary garbage.

So what locations have what on the Apple II? The 6502 programmer has to know, lest he try to store his data in ROM. The memory map is a useful tool for seeing at a glance the basic layout of the 64K addressing range.

Apple II Memory Map

PAGE NUMBER

DEC	HEX	FUNCTION
255	FF	
.		
.		ROM (12K; $D000 - $FFFF)
.		
.		
208	DO	
207	CF	I/O (4K; $C000 - $CFFF)
192	C0	
191	BF	
.		
.		
.		
.		
.		
.		
.		
.		
.		RAM (48K; $0000 - $BFFF)
.		
.		
.		
.		
.		
.		
.		
.		
.		
.		
.		
0	00	

Three fourths of the addressing space is devoted to RAM. In the early days (1977-79), most Apples rolled off the assembly line with only the lower 16K of memory installed. (Or even only 4K–fancy that.) The machines weren't any different; RAM chips just used to cost more than $1 apiece and down (like $30 each). If you tried to write a number to location $4010 in a 16K Apple II, there was nothing to stop you from trying–but it wouldn't save your number. Nowadays 48K is just about universal. This relatively large amount of RAM (in 1977, people **killed** for 48K) gives the Apple a lot of flexibility, as you are free to do anything you want in RAM, from Pascal to Basic to graphics to programs that impersonate microprocessors.

$C000-$CFFF are 4,096 locations devoted to I/O. Without these locations, the 6502 could not share any of the marvelous things it can do with humankind. These addresses are connected to Apple hardware, like the speaker, keyboard, game paddle connector, and disk drive.

From D000-FFFF is ROM. Stored in ROM is a 6502 machine language program that runs programs called Applesoft, and a series of utility routines that take care of reading keyboards, displaying text, inspecting game paddle controllers, and so on, collectively referred to as the Apple monitor.

MASS STORAGE

$BFFF bytes is a lot of RAM–but sometimes not enough. Enter the Disk II mass storage unit. Here's a device that can store 140,000 eight bit numbers on one disk–and we can have as many individual disks as we can cope with. (For me, 10 is that number –after the tenth disk all catalogs start looking the same.) On occasion, the 6502 instructs the disk drive to load some of its contents into RAM. Once in RAM the 6502 can deal with the bytes in the normal, intimate, fast way. Disks also have the very useful property of not losing their numbers when power is removed.

BUT HOW DOES IT WORK?

The movie **TRON** ("I'm going to put you on the game grid, Flynn") notwithstanding, the world of the 6502 is as far removed from human experience as anything could possibly be, more like the whirling cams and levers of a bottle capping machine than men in funny hats playing catch with luminous Frisbees. Even so, an analogy relating the 6502 to the actions of human beings is the best way to explain how machine language works.

Consider, if you will, the loading dock of Giant Metropolitan Software Publishing House, Inc. Delivery trucks move ponderously in and out of loading bays. Workers with dollies and fork lifts move refrigerator-sized cartons of blank disks coming in and completed programs and manuals going out.

The undisputed boss of the dock is the foreman. An imposing figure in sky blue jump suit and orange Astros cap, directing workmen to and fro, signing paperwork, glancing occasionally at a clipboard in his left hand.

He runs things tight, by the Book. The Book is a much worn spiral notebook of maybe 150 pages chained to his desk. The label on the torn cover, although now illegible from years of use, once said: "SHIPPING DOCK PROCEDURES MANUAL". Each page is numbered. Some pages have only two or three lines on them, others, 10 or 15. Page 12, for example, says, in careful lettering:

JOB 12 UPB - UPS BLUE SHIPMENT

STEPS:

1. PACKING LIST FOLLOWS WORK ORDER.
2. FILL OUT UPS LABEL
3. LEAVE AT UPS AREA
4. DONE

Every morning the foreman finds waiting in his IN basket a stack of GMSPC, Inc. workorders. A workorder has some inter-office mumbo jumbo on it, and, in the upper left hand corner, the all important shipping dock procedure number. Not all of the sheets in the stack are work-orders; most of the workorders need the sheet or two of paperwork with them to be complete.

The basket stays full all day long, with clerks periodically replen-ishing it. The dock foreman's most important tool is his green work-order clipboard. After his morning coffee he takes the first work-order from the stack and clips it to the clipboard. As long as that workorder is on the clipboard, he will devote his energies totally to performing the operations required to fulfill it.

There's a bunch of writing on each workorder, but he's only interested in the number in the upper left hand corner. The procedure number. Today's first workorder has a procedure number of 22. "TPC", he

mumbles to himself as he flips to page 22 of the procedure manual. (He knows 22 by heart, but turns to the page anyway. He's that kind of man.)

PROCEDURE 22 TPC - TEXPAK C.O.D

STEPS:

1. PACKING LIST FOLLOWS WORK ORDER.
2. CALL TEXPACK FOR PICKUP
3. FILL OUT C.O.D. LABEL. COMPANY CHECK OKAY
4. MOVE PACKAGE TO SHIPPING AREA
5. DONE

When he finishes the last step of TPP, if it takes 5 minutes, or 15, he comes back to his desk, unclips the old workorder and puts it and the packing list that went with it face down in the OUT basket. Without a pause, he takes the next workorder from the In box, tacks it to the clipboard, and goes to work on it. All day long: Get a workorder. Look up procedure. Perform the workorder. Get a workorder. Look up procedure. . .

A 6502 runs the same way: Access a memory location. Decode the contents of that location. The instruction may require the next byte or two in memory for execution. Execute the command, and proceed to the next memory location for the next instruction.

THE FANTASTIC VOYAGE

Remember the movie **Fantastic Voyage**? Where some intrepid scientist/military types are shrunk to the size of a microbe to assist in the removal of a tumor from a valuable (I **guess!**) scientist's brain?

Through the magic of the printed word, we're going to do the same thing. Only without Raquel Welch in the crew. Take that back–she can come too. We climb into our manta ray shaped submarine and buckle up. Brace yourself. Soldiers are blasting us with strange violet light. We're shrinking. We're getting smaller. . .smaller. . .smaller. . .

We're so tiny now that the dot that ends this sentence looks like the Astrodome. We lift off (this submarine can fly, too) and head straight for a nearby Apple II. It looks as big as Mount St. Helens. We slip easily through a crack in the keyboard, into a bizarre, alien landscape of ribbon cables and clock crystals. After thirty minutes of steady cruising, suddenly, dead ahead is a huge black monolith. The objective of our mission: the **6502 microprocessor**. . .

4.
Getting Started

It's time to get acquainted with The Visible Computer. Take the TVC disk from its envelope, slip it into drive one, and power up. Apple II Standard (instead of Plus) owners will need to do an intermediate step: Boot the computer on the DOS 3.3 System Master to load Applesoft into a RAM or language card. Insert the TVC disk, and boot it with a PR# 6.

In a few seconds, you'll see a Software Masters (tm) copyright message that will remain onscreen for several seconds, or until a key is pressed, whichever comes first. If you don't see this message, we've got problems.

IF THE DISK WON'T BOOT

Is your Apple 48K? Does it have a 16 sector (DOS 3.3) controller card? Is it an Apple II Plus? Or, if not a Plus, does it have either an Applesoft Rom card or one of the many 16K RAM cards (Apple Language Card, Microsoft RAMcard, etc.)? If your answer to any of these questions is no, you'll have to correct the situation before TVC can run on your machine.

If your system meets these requirements and still won't boot, you may have a bad disk. See your dealer, or contact Software Masters at the address on the back cover of this manual.

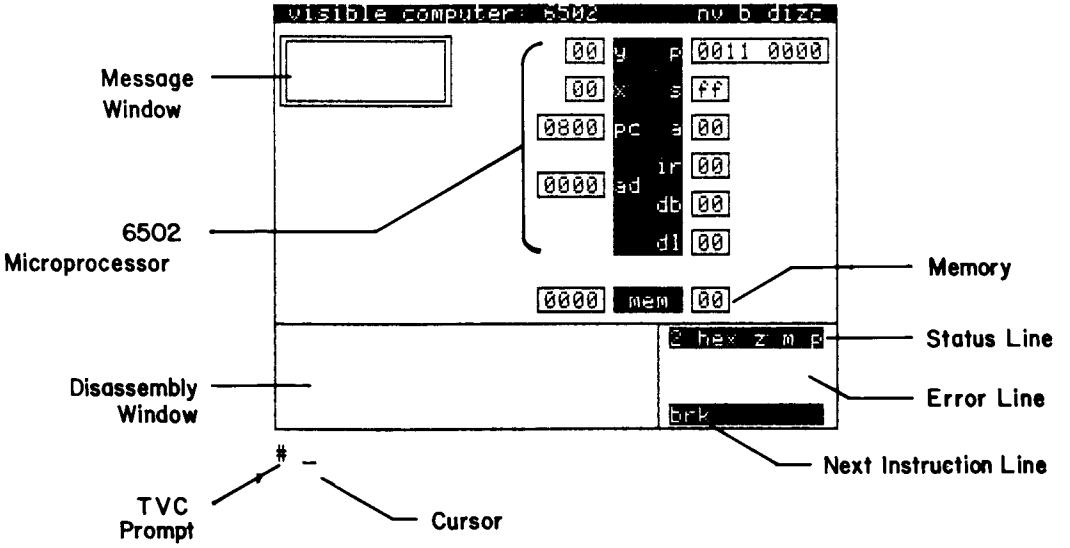
The copyright display is replaced by the message:

LOADING...

TVC is a big chunk of machine language and binary data, and an even bigger chunk of Applesoft Basic. Loading it all takes about 15 seconds. When it finishes you'll get the message:

INITIALIZING...

And in a second or two the TVC display appears.



Visible Computer Display

You're now looking at something that few outside of the halls of Intel, Motorola, and the ilk have ever seen; the innards of a working microprocessor.

Each of the boxes holding a hex number is a **register.** A register is just a place inside the processor where we can store binary numbers, a lot like a memory location. The 6502 can perform marvelous feats with a few simple operations on the contents of these 10 registers. In the next chapter we'll begin to see how this is done.

Speaking of memory, the 6502's contact with the 65,536 address locations is via the two registers of the separate area labeled **mem** (memory). The 16 bit register is for the address; the eight bit register is for the data stored at that address. During program execution, this is where numbers appear that are being stored in and retrieved from memory.

The message window in the upper left hand corner is where TVC outlines the steps it follows in executing each of the 151 opcodes of the 6502 instruction set. When TVC is not actively running a program (now, for instance), this window is blank.

We'll put off talking about the disassembly window until we learn what disassembly is. The TVC status window displays various tidbits germane to TVC's execution. At the very bottom is the all important command line, where you'll enter commands to control TVC. The "#" (pound sign) is the TVC monitor prompt. Like the Applesoft prompt, it serves as a reminder that entries should be statements recognizable by TVC.

The blinking line next to the prompt is the TVC cursor, and, like the flashing block cursor of Applesoft, shows you where you are on the screen when typing. A cursor is one of those overlooked things in life that you don't really appreciate until you haven't got one anymore.

The Visible Computer consists of two major parts: the **monitor** and the **6502 simulator.** The monitor controls ("monitors") the simulator. The simulator is the part that actually executes 6502 programs. Throughout this manual we will use phrases like "in the monitor" and "returning to the monitor." You are "in the monitor" when the prompt is at the bottom of the display. You are "in the simulator" whenever the prompt is **not** at the bottom of the screen.

MONITOR COMMANDS

TVC has a 21 command vocabulary. You control TVC by typing instructions at the monitor prompt, something like conversing with an adventure game. You tell it something, and, if what you told it is something it understands how to do, it'll do it.

"GET ROCK"

"SORRY, I DON'T KNOW THAT WORD"

GENERAL RULES FOR ENTERING TVC COMMANDS

To issue a command, type your request and press return. If a command consists of more than one part, use spaces between the parts to separate them. One is sufficient.

If you make a mistake in typing a command, correct it by either using the back-arrow key and retyping, or by typing a Ctrl-X, and starting from scratch. If TVC cannot understand your instruction, it will tell you so with error messages.

Commands have the general form:

COMMAND [argument1] [argument2]

Argument is a 25 dollar computer word that means "modifier". Some TVC commands need no arguments; others need additional information to be complete. Just as some Applesoft commands ("HOME", "NEW") stand alone, while others ("IF", "GOSUB") don't make sense unless you include more information.

Examples of one word monitor commands are ERASE and RESTORE.

The monitor command BASE (change a register's base to hex, binary, or decimal) needs two arguments; one to indicate the thing we're changing the base of, and another to specify the new base. BASE PC BIN changes the display mode of the program counter to binary.

You **must** separate a command and its arguments by one or more spaces. You **must not** use spaces **within** a command or argument. For example, if you entered the ERASE command as ER ASE, the command interpreter of TVC would understand it to mean: "Perform ER using argument ASE". Which, upon trying to find command ER, produces an error. If you can't get a command to work, check your syntax–and don't forget the spaces.

Now to get our feet wet with a couple of commands. First, we'll call up the calculator and see if two plus two equals four. I know that's a question of great concern to many of you.

Type "CALC" and press return. If you make a mistake, fix it with back arrows, Ctrl-X's, and retyping. If "Command" appears in the TVC status window, accompanied by a low beep, TVC is telling you it cannot understand what you entered.

Eventually you should see the following on the command line:

<HEX><CALC> 00

The cursor will be positioned under the first zero.

The "HEX" tells you that the current calculator base is hex. This means that all numbers produced by the calculator will be displayed in hex (**without** dollar signs), and that the numbers you enter must contain only characters valid in hex. In other words, no hex numbers like G3#B, or decimal numbers like FC3. Do not include dollar signs; if the calculator's base is hex, the dollar sign is understood. The calculator base can be changed by typing a Ctrl-B for binary, and a Ctrl-D for decimal. For now leave it in hex (Ctrl-H).

Enter: **2 + 2 <return>.**

About one second after you hit return, the **2 + 2** is replaced by **04.** If you didn't get four for an answer, make sure you include the spaces between the two's and the plus sign.

Try:
3 + 3
4 * 4
6 / 2
3EA * C. (Try that on your Casio!)

To use the calculator to convert between bases, follow these steps. Converting 65,000 decimal to hex:

Ctrl-D
65000 <return>
Ctrl-H.

Convert it to binary with Ctrl-B, and back to decimal with Ctrl-D. With the base decimal, try adding FF to 3. The BASE error that results is TVC telling you that you have entered characters not valid in the current base. FF is not a valid decimal number.

Answers are displayed with leading zeros, and for binary numbers, with spaces separating each nibble. How many total characters are displayed is a result of the size of the number. Numbers less than 256 will always display as two, three, or eight digits (for hex, decimal, and binary modes, respectively). Numbers greater than or equal to 256 display as four, five, or 16 digits. This is a by-product of the calculator's use of the same display and conversion routines used elsewhere by TVC.

This calculator has certain properties that make it undesireable for everyday checkbook balancing and miles per gallon calculations. First, it is an **integer** calculator. Numbers with decimal points are not allowed as input. Divisions produce truncated (chopped off, as opposed to rounded off) results. (e.g., **6 / 2 = 3. 5 / 2 = 2. 9 / 10 = 0**).

You may not enter negative numbers. Dashes entered anyplace except as the operator are treated as invalid characters. If you do a subtraction that produces a negative number, by subtracting a larger number from a smaller number, the answer will be displayed in **two's complement** form (later we'll learn what that is). Lastly, you may not enter, or produce via calculations, numbers greater than 65535. These quirks are a result of the calculator's purpose in life: To help you write machine language programs.

When you've had all the fun you can stand changing numbers back and forth between bases, return to the monitor by typing escape. Got the monitor prompt back? Good. Next, try this short and sweet command:

ERASE

Wow. Spectacular. This command clears a space where you can experiment with high resolution graphics. But since it's sad to see a lonesome little monitor prompt all by itself, bring the display back with the **RESTORE** command. If you like, you can issue these two commands over and over. For the more adventurous, let's move on.

Type **WINDOW OPEN**. Now we're erasing only a part of the display. When you know more about 6502 programming, you'll appreciate the choice of registers that remain onscreen when the window is open. Again, we don't want to leave our display looking so empty, so replace the part that got erased with WINDOW CLOSE.

What's so great about lower case? Anyone out there with a grudge against lower case? If so, get rid of those pests with the command CASE UPPER. Although it may take a little getting used to, "Experts"

have proven that people comprehend lower case letters more quickly than upper case. Now that you're convinced of the superiority of lower case, change back with CASE LOWER.

What's so great about hex? TVC defaults to a display mode of hex for all registers except P, but you don't have to leave it that way. Change the base of all the registers to binary with:

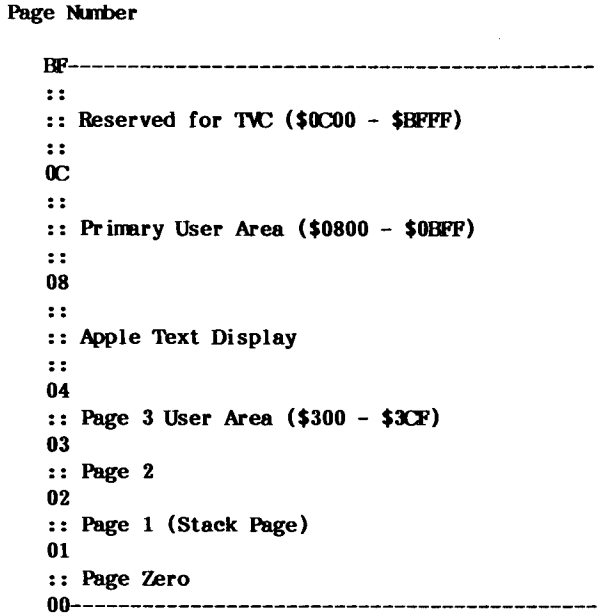
BASE ALL BIN

Change only the A register to decimal with **BASE A DEC.** You can mix and match any combination of hex, binary and decimal. Get it looking like you want a 6502 to look.

This concludes our first session with TVC. We've learned what the monitor is, and experimented with the commands CALC, ERASE, RESTORE, WINDOW, and BASE. In the next chapter we'll go in up to our knees and splash around a little.

5.
Working with Memory

TVC Memory Allocation



In this chapter we'll learn how TVC subdivides the Apple's 48K of RAM. Then we'll practice the monitor techniques of examining and changing the contents of memory.

Although you can **read** bytes from almost anywhere in memory, The range $E00 - $BFFF is offlimits to **writes.** If you were allowed to populate this region with the numbers of your choice, you might hurt TVC; maybe crash it, maybe just subtly alter a single function. TVC will appear to accept an order to place a value at $4003 (no error messages), but not obey it. It handles ROM and I/O locations the same way. Later, when you've proved to be a responsible person, you'll learn a command that lets you to write to these areas.

The 2K that is available for writes ($0000 - $3CF and $800 - $BFF) is plenty of room for most machine language programs. The area from $400 to $7FF labeled "Apple Text Display" should not be used, although you won't hurt TVC by doing so. You may not find the numbers you stored there when you come back for them later.

The monitor provides three methods of getting numbers into and out of these locations. A fourth way is to write a program that does the work for you–but that comes later.

A WINDOW INTO MEMORY

To display the contents of 16 memory locations at once, use the **WINDOW MEM** command. Unless you tell it otherwise with the LC or RC commands, TVC displays locations $800-$807 and $0000 - $0007 in the memory window. You can change the base of the memory window to decimal with: **BASE MEM DEC. BASE MEM HEX** changes it back. If you want to change the value of one of these locations, there are three options.

DIRECT LOAD METHOD

The quickest way to write to a memory location is a direct load. Enter the address and the value you want stored at that address separated by a space. Both address and value must be numbers valid in the current monitor base (the second entry on the TVC status line–hex, if you haven't changed it since booting). To put $CA in location $806, enter:

806 CA.

See the contents of location $806 change? How nice. **804 3** writes a three into location $804. To use decimal numbers, set the monitor base to decimal with BASE MON DEC. Now addresses and data must be given in decimal. All the examples in this book will use hex, so change it back.

Direct loads are okay for a couple of quick writes, but if we want to write data to 50 consecutive locations, it's a lot of work to specify the address each time. A more efficient way to change several consecutive locations is the EDIT function. Invoke editing with the command:

EDIT 0

This causes editing to commence with memory location $0000. To change the contents of location 0, enter a number (naturally, an 8 bit number valid in the current monitor base). The number you enter replaces the

previous value, in memory and onscreen. The address is incremented by one and the process repeats. There are a couple of tricks you can accomplish with your first keystroke. A back arrow displays the contents of the **previous** location. A front arrow jumps to the **next** location, without changing the number stored at the first address. Escape returns you to the monitor.

If you write to a valid location not displayed in the window, the change is made in memory, but not onscreen. There are two commands to change what memory locations are displayed, **LC** (change left column) and **RC** (change right column).

To display locations $100-$107, enter:

LC 100

If you want, you can display the same locations in the right column. It's a free country.

LOADING FROM DISK

The BLOAD command loads memory from data stored in DOS 3.3 B-files. The demonstration 6502 machine language programs we will be using shortly are loaded this way. You can also use BLOAD as a handy way to write zeros into your working area, to clean it up. This is done by BLOADing a file on the TVC disk consisting of nothing but zeros, named, appropriately enough, ZEROS. Try it now.

BLOAD ZEROS

This zeros all the bytes from $800-$BFF, the main working area of TVC, as well as pages zero and one.

As you might imagine, there is a counterpart to BLOAD named BSAVE. BSAVE writes a selected area of memory to the file of your choice. We will not be using BSAVE for awhile; in fact, until you have passed a couple of milestones in your machine language studies, you will not be allowed to use it. Don't believe me? Try the command:

BSAVE TRYANDSTOPME

Until you're a TVC master, no BSAVEs.

CHANGING REGISTERS

Earlier we learned how to change the display mode of a register. There's also a way to change a register's contents. Next to the monitor prompt, enter the name of the register and the value you want to put there. As with all monitor commands, express the value in the current monitor base. To place $89F in PC, enter:

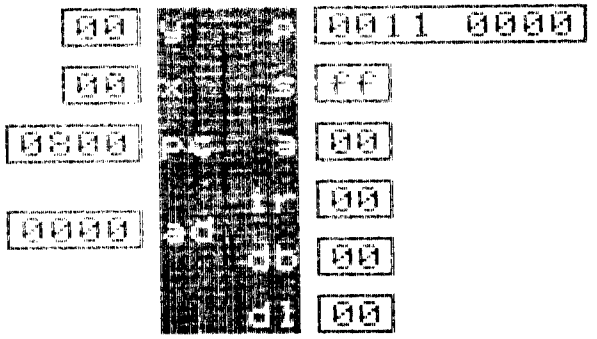
PC 89F

You may not write numbers larger than 255 into an eight bit register, or larger than 65-you-know-what to a 16 bit register. Practice with it. Change the base of registers you've written numbers to. Do you get the same conversions you get on paper or with the calculator?

Appendix C is a reference on all 21 TVC commands. Even though there are some that we won't be using for some time, turn there now and quickly look through it.

6.
First Programs

The 6502 you are (or should be) looking at is one that trades speed for user friendliness. In exchange for being about a million times slower than a real 6502, it allows you to peek inside as it runs.



A TOUR OF THE 6502

The 6502 has eight 8 bit registers. A register, remember, is just a box where we can put binary numbers. Their abbreviated names: A, S, P, X, Y, DL, DB, and IR. There are two 16 bit registers, PC and AD. We'll discuss each register individually, as they have more personality than the typical memory location.

A The A register, or accumulator, although not especially large, is probably the most important register in a 6502. It gets a workout in almost every program.

S Directly above it is S, the "stack pointer" register. S is used for stack operations.

P The P register (Processor Status) holds the distinction of having probably the most unnatural abbreviation of all 6502 registers. Don't blame me. It also is the only one that defaults to a binary display, because we are more interested in P's individual bits than their collective value.

X
Y

Next are the ever-popular X and Y registers. These two get a lot of use, but not as much as A. They are often called **index** registers because of their use in something called indexed addressing.

PC

The big register beneath X is the program counter. It serves as a placemark to remind the 6502 where it is in memory and what instruction it should execute next. Fans of the program counter could make a good case for it being the most important register in the 6502. At the very least it's twice as big as the accumulator.

DL

DL, the data latch, is the 6502's bus station, the crossroads of data coming into and out of the processor to and from memory. No data comes into or leaves the 6502 without passing through this register.

DB

DB, for data buffer, is a place where we can temporarily shuffle a number off in the middle of an instruction until we're ready for it.

IR

IR is the instruction register. This is where a 6502 deposits the instruction that it is currently being executed, It's the 6502's equivalent of the dock foreman's workorder clipboard, a place where an instruction can be studied ("decoded") to figure out what it is and how to execute it.

AD

AD is the address latch (sometimes called address bus), the place that holds the memory location to be accessed during reads and writes.

The 6502's 16 bit registers, AD and PC, have something of a dual personality; sometimes they behave like one big 16 bit register, other times like a pair of 8 bit registers. When used in this way, the high order halves are called PCH and ADH, and the low order halves, PCL and ADL.

A, S, P, X, Y, and **PC** are sacred abbreviations agreed on by all 6502 programmers. The other registers, **DL, DB, IR,** and **AD** have more flexible names, as they were invented by the author of this manual. That's right. You could buy 11 books on 6502 machine language, and not one would mention the DL, DB, IR, or AD registers. The reason is that a programmer does not have to worry about the contents of these registers to write 6502 programs. They're in every 6502, essential to the running of things, but since they perform temporary, scratch pad functions, the programmer need not concern himself with them. Since

TVC simulates the inner workings of a 6502, we couldn't leave them out.
Now to put some of this knowledge into action. Let's load and execute the first of the demonstration programs on the TVC disk, named, appropriately enough, PROG1. Load PROG1 (no spaces between the "G" and the "1") into memory, all two bytes of it, with the command:

BLOAD PROG1

Unless you specify otherwise, BLOAD loads data starting at address $800, so PROG1 now resides in RAM beginning at $800. PROG1 is a simple affair that will accomplish one small feat. It will cause a $33 (51 decimal, 0011 0011 binary) to appear in the accumulator. I know you could easily do that with the monitor command: **A 33**, but bear with me.

Let's look at the data that makes up PROG1. Put the window in memory mode with the WINDOW MEM command. PROG1 consists of the $A9 at $800 and the $33 at $801. Hmmmm. . . He said the program was going to put a $33 in the accumulator and one of the two bytes in the program is a $33. Could be a connection.

The zeros that follow mark PROG1's end. Not a very complicated (or useful!) program. CLOSE the WINDOW. We want to see the whole processor-memory setup for our first program. Next, put TVC in its slowest, most helpful state with the command:

STEP 3

The current step value is the leftmost item on the status line. If it doesn't say three yet, get with it.

I know you're anxious to get started, but before we turn the simulator loose on PROG1, consider the current contents of the registers. Since we just booted TVC, the registers are in their default condition. Most, but not all, hold zeros. For now, don't worry about poorly abbreviated P and its binary contents, or the $FF in the stack pointer. I call your attention rather to the $800 stored in the program counter.

This $800 is where in memory the 6502 that's about to come to life will find the instruction it will execute. It is no coincidence that BLOAD placed PROG1 at $800. If we were to make the program counter something other than $800, say $1AFF, the simulator would not execute PROG1, but rather whatever unknown data it found laying around at $1AFF.

To enter simulator mode, press return without entering anything at the monitor prompt. Things happen fast now, so pay attention.

SIMULATOR MODE

This is our first excursion into the 6502 simulator. It does not have nearly as many commands to worry about. It executes programs while you watch. The message window illuminates and displays "FETCH" (written in computer-style print; all the better to signify that this is a 6502's thoughts we're seeing here). This means an instruction fetch, the first phase of executing an instruction, is in progress. When the 6502 has fetched a byte and placed it in the instruction register, the fetch cycle ends, and the execution phase begins.

The other line of the message window contains a more cryptic message.

T: PC -> AD

This translates into English as **"Transfer: The contents of the Program Counter to the Address Latch"**. This is a **micro-step**, one of eight known by the TVC simulator. A microstep is to a 6502 instruction as a proton is to an atom; instructions are built by combining eight basic microsteps in a specific order. The Visible Computer microsteps are listed in Appendix D. The transfer microstep, which blinks the source and then the destination register, is the most common, used by every instruction at least twice.

The transfer will occur as soon as you exit the pause you are currently stuck in. A pause can be ended by any key except "C". "C" invokes the calculator, the only monitor function available from within the simulator. Exiting the calculator with escape returns you to the pause.

Tap the spacebar and proceed. AD now contains $0800; note that PC is still $800. A transfer doesn't affect the contents of the source register.

READ is the next microstep of the FETCH process. A read happens fast, so be ready. It consists of the following steps:

> **The contents of the address latch are transferred to memory's address bus.**

Memory fetches the contents of that address and transfers it to memory's data latch.

That value is transmitted to the 6502's data latch.

Before we let this READ happen, a pop quiz: What value will be read from location $800? Answer: $A9. It won't have changed from a minute ago when we looked at it from the monitor. Okay, press the spacebar.

The 6502 is still in the fetch cycle. It's taken a workorder from the In-box, but hasn't got it to the clipboard yet. Until it gets this byte to the instruction register (IR), the 6502 doesn't have any idea of what the instruction is, much less how to complete it. The next step, then, is to put the instruction in IR so we can get on with decoding and executing it. As soon as the $A9 is in IR, the **fetch** phase ends, and the **execute** phase begins. "FETCH" is replaced in the message window by "LDA IMMED", the 6502 saying to itself "I need to load my accumulator with the next byte in memory." A Load Accumulator, Immediate, also known as instruction number $A9.

And it knows what to do next. First step: Increment the program counter. Now it contains $801. Transfer it to the address bus. Do you feel a READ coming on? Yes. Read the contents of location $801 into the data latch. Copy the number you found there, a $33 (big surprise), into the accumulator.

Almost done. All that remains are a couple of details. Something called "CONDition FLAGS" happens that blinks the P register. More on this phenomenom later. And a closing increment of the program counter. We do this not to assist in the execution of this instruction, but to **prepare for the next one.** When you're in the monitor, the program counter always points to the next instruction, not the end of the one just completed.

A real 6502 doesn't have the luxury of sitting around doing nothing while a monitor takes over for half an hour. It has to execute one instruction after another, bing-bing-bing, hundreds of thousands of times a second, without so much as a break to pat itself on the back. So every instruction sets the program counter to point to the desired starting point of the **next instruction.**

That last increment of the program counter completed the instruction, and deposited us in the monitor, and if you haven't got too quick a finger on the return key you're still there. The last act of the simulator is to list the instruction just performed in the disassembly window. Understanding the exact format is not important right now. Consider the disassembly window a trail of the last five instructions

the simulator has executed.

What would happen if we were to enter the simulator now? Try it. It'll fetch the fetch the $00 that's in $802, put it in IR and digest it. $00 is a 6502 instruction called BRK (software break). BRK puts you right back in the monitor without doing anything. It's a signal to the simulator that the program is over and we want to get back to the monitor. Later we will learn more about this unique instruction.

Congratulations! You have just watched your first program. If you were able to follow along, you have learned about 90% of the fundamental basis of machine language. Before you go on to the next session and progressively more complex programs, make sure you understand how this one works. You can have an instant replay of PROG1 by setting the program counter back to $800 with **PC 800.** While you're at it, why don't you change the contents of memory location $801 from $33, to say, your age–then PROG1 can serve the useful purpose of telling the accumulator how old you are.

MOVING RIGHT ALONG

So far we know exactly two of the fifty-six 6502 instructions. LDA, also known as $A9: "load the accumulator with the byte following this one", and BRK, $00, "Break out of the program and return to the monitor". "LDA" and "BRK" are not haphazardly chosen abbreviations; they are official 6502 **mnemonics** (neh- **mon**-ics). A mnemonic is a memory aid, based on the theory that it's easier for human beings to associate "LDA" with the act of loading the accumulator than $A9. The 6502 has no idea, of course, what LDA means; if you want a 6502 to load its accumulator you have to give it the **opcode** $A9. Each 6502 instruction has a three letter mnemonic. Some of the abbreviations are better than others, but all are easier to remember than a number.

Remember me saying that the accumulator is the most important register on the 6502? That makes LDA-$A9 a good instruction to know. Loading is all well and good, but what about **storing** a value in the accumulator somewhere in memory? Is there a way to do that? You bet. BLOAD PROG2.

PROG2 introduces the flip side of LDA, STA (Store Accumulator; opcode $85). This instruction causes the contents of the accumulator to be placed in the memory location of our choice. PROG2 will first LDA with $66, and then STA it at memory location $43 (a page zero address). PROG2 is longer than PROG1, a whopping 4 bytes. Take a look at it with either WINDOW MEM or EDIT. It begins, as did PROG1, at $800.

Notice the $43 at $803. A coincidence? You know better.

With PC set to $800 to start PROG2 at the beginning, and with WINDOW CLOSEd and STEP 3 in effect, step your way through this two instruction program. Pay close attention to STA-$85. STA is a tad more complex than LDA-$A9.

When the 6502 sees an $85 in the clipboard register, it knows it must get one more byte out of memory, just as it did with LDA. But what it **does** with the second byte (a $43) is different.

First it transfers it to the address bus. Since AD is 16 bits wide, and we're loading it with an eight bit number, the most significant byte becomes zero. We have now formed the zero page address $0043. Putting a number on the processor's address bus is always a precursor to reads and writes of memory. Next, we transfer the accumulator to the crossroads register, DL. The stage is now set for the WRITE microstep.

A write consists of the following steps:

> **The contents of the address bus are transferred to memory's address bus.**
>
> **The contents of the data latch are sent to memory's data latch.**
>
> **Memory inserts the value into the selected location.**

After the write, STA is complete except for a final increment of the program counter to make it point to the next instruction. No flag conditioning this time.

When you get back to the monitor, check the contents of memory location $0043 with either WINDOW RAM (and an LC) or EDIT and verify that it really got the value PROG2 put there. Run this program several times. Use different values for $801 and $803. DO NOT change the opcode values, the $A9 in $800 or the $85 in $802. Change them and you change the instruction from LDA to who knows what.

That's three instructions down, 53 to go. But we're about to learn 10 more with astounding ease.

LOADING AND STORING SOMETHING BESIDES THE ACCUMULATOR

The accumulator is top dog on the 6502, but once in a while we need to load and store some of the other registers, too. There are instructions for just that. LDY and STY for the Y register. LDX and STX for X.

MNEMONIC	OPCODE	OPERATION
LDX	$A2	Load X register
LDY	$A0	Load Y register
STX	$86	Store X register
STY	$84	Store Y register

PROG3 demonstrates all the instructions we've learned. BLOAD it, set PC to $800 and step through it. Each of the new instructions functions exactly like its accumulator counterpart. We're really starting to accomplish things with PROG3; three consecutive memory locations loaded with $FF. Great.

All of the instructions so far have been two-byters: An opcode byte to give the 6502 its orders, and a second byte to use in completing the order. The 6502 has one byte instructions, too. Instructions that are so self explanatory they don't need a second byte to finish the job. Such instructions are said to be "implicit", or **implied.**

The six Transfer instructions are representative of the Implied group of 6502 instructions. They are used to transfer the contents of the X and Y registers with the A register, and between X and the stack pointer (S). In table form:

MNEMONIC	OPCODE	OPERATION
TAX	$AA	Transfer A to X
TAY	$A8	Transfer A to Y
TXA	$8A	Transfer X to A
TYA	$98	Transfer Y to A
TXS	$9A	Transfer X to stack pointer
TSX	$BA	Transfer stack pointer to X

Don't confuse the 6502 transfer instructions with the "T:" microstep. A "T:" is one phase of execution of a 6502 transfer instruction—in fact, of every 6502 instruction.

Two of the transfer series are demonstrated in PROG4. For now, take the others on faith. BLOAD and STEP 3 your way through it. Notice that the 6502 knows it need not fetch any additional bytes out of memory after the instruction fetch to complete a transfer instruction. It knows what to transfer where by looking at the opcode byte.

Notice that the 6502 knows it need not fetch any additional bytes out of memory after the instruction fetch to complete a Transfer instruction. It knows what to transfer where just by looking at the instruction.

Ultimately, PROG4 accomplishes the same function as PROG3 (the not-so-earth-shaking feat of writing $FF into locations $40, $41 and $42), but does it faster. It takes less time to execute a one byte transfer instruction than a two byte load instruction. It's also two bytes shorter.

Now we're making some progress; 13 instructions down, 43 to go.

7.
Processor Status Register

The P (processor status) register is something of an oddity in the 6502 family. Not only does it have a confusing abbreviation, it is also the only register where we are more interested in contents on a bit rather than byte level. In other words, if both P and A happen to contain 0011 0011, we will usually interpret A as containing the number $33, and P as containing ones in positions 0, 1, 4, and 5, and zeros in positions 2, 3, 6, and 7. P defaults to binary display so that each bit falls under its abbreviation.

Speaking of defaults, why are two bits set? Because that's what you usually find in this register inside a real 6502 running in an Apple II. The full names of these rugged individualists:

N Negative flag.
V oVerflow flag
B Break flag
D Decimal Mode flag
I Interrupt Disable flag
Z Zero flag
C Carry flag

Two things: "Flag" is a fancy term for bits of unusual importance. Bit 5 of the processor status register is not used. It's **there**, obviously, but we have no control over it, nor will we ever be interested in its value.

INSTRUCTIONS THAT AFFECT THE P REGISTER

There are implied (one byte) instructions to set and clear many, though not all, of the P register flags. (**Set** and **Clear** are handy verbs describing the act of forcing a bit to become either a one or a zero, respectively. "Reset" is used interchangeably with "clear" in this manual).

COMMANDS THAT CLEAR P REGISTER BITS

MNEMONIC	OPCODE	OPERATION
CLC	$18	Clear carry flag
CLD	$D8	Clear decimal mode indicator
CLI	$58	Clear interrupt disable indicator
CLV	$B8	Clear overflow flag

COMMANDS THAT SET P REGISTER BITS

MNEMONIC	OPCODE	OPERATION
SEC	$38	Set carry flag
SED	$F8	Set decimal mode indicator
SEI	$78	Set interrupt disable indicator

This list of commands is incomplete; there are no instructions for setting or clearing the negative and zero bits, and none for setting overflow. There don't need to be, as we shall see.

THE ZERO FLAG: 6502 HISTORIAN

The Z flag contains a single binary fact about previously executed instructions. It is "conditioned" (set or cleared) by the 6502 every time it executes a load or transfer instruction. If you load a zero into the accumulator, Z will be set. This is backwards from common sense, so I repeat: If you load X, Y, or A with a zero ($00; 0000 0000), the Z bit will be set. It will **stay** set until such time as another load or transfer comes along that loads a **non-zero** value into a register. Once cleared, it will stay that way until the next zero load comes along to set it.

THE NEGATIVE FLAG

The N flag is also conditioned with every load and transfer instruction. If you load or transfer a number that has bit 7, the most significant bit, set, N will be set. Any 8 bit number greater than $7F has this bit set (check it out!). Conversely, loading or transferring values with this bit clear will clear the N flag. N gets its name from the fact that frequently bit 7 is used by the programmer to indicate negative numbers. We will describe the signed number situation in more detail later on, but quickly, the convention is: If bit

7 is set, the number is negative. If it is reset, the number is positive. If you are not using signed numbers, the behavior of the N flag can be disregarded.

This conditioning effect, in conjunction with instructions to be presented in the next chapter, allow the programmer to test conditions that existed on previous load and transfer instructions. The technique is to examine the state of the Z or N bits, and decide what to do next on the basis of that finding. This is related to Basic's IF ( ) THEN GOTO statement.

BASIC

```
IF A=0 THEN GOTO 1000
A = A + 1
etc.
```

Machine Language

```
TXA
[If accum = 0, Jump to XXXX]
```

In the next chapter we'll learn an instruction to fill in the brackets.

PROG5 demonstrates both the implied clear / set instructions and the conditioning effect of loads and transfers. BLOAD PROG5, but before you run it, we're going to explore a feature of TVC for **anticipating** what a program will do without actually running it.

DISASSEMBLY: THE L(ist) COMMAND

With PROG5 BLOADed, type: **800 L.** As with all monitor commands, separate the L from the 800 with a space. What appears in the disassembly window is a sneak preview of the first five instructions in PROG5. Unlike the instructions put there by the simulator after executing an instruction, the address is not shown in inverse video.

Disassembly is an awkward word for the extremely useful process of presenting a machine language program in a form more palatable than plain hex. The hex is there, address and contents–but the humanized version of the instruction is what we're really after. Disassembling a machine language program is not the same as executing a program, any more than listing a Basic program is the same as running it.

A disassembled instruction contains two parts; **Mnemonic** and **Operand.** In "LDA #$33", **LDA** is the mnemonic, and **#$33** the operand. Both assist, sometimes subtly, the programmer in determining what the instruction does.

The "next instruction line" of the TVC status area, if you haven't already guessed, holds the disassembled form of the instruction that is either about to be executed (if you are in the monitor) or is currently being executed (if you are in the simulator). Minus the address (which is defined to be the program counter, anyway, and the hex values themselves. The next instruction display changes whenever the program counter or memory pointed at by the program counter is changed.

Although now PROG5 will be anticlimactic, having already seen what instructions are in it, work your way through it with the simulator. People with printers can have a little extra fun by activating the output-disassembly-to-printer feature of TVC with the command:

PRINTER ON

The Set/Clear instructions are straightforward enough, but pay special attention to the COND FLAGS microstep of the loads and stores that follow. Each load conditions the N and Z flags. If we load a register with a zero, then the 6502 will set Z. If it was already set, it'll stay set. N is altered at the same moment. It will be set whenever a load occurs that sets bit 7 of the register that is loaded, and reset when bit 7 is not. For now, just observe the conditioning process and don't worry about why it goes to this trouble.

Tinker around with the data portion of the load instructions. What do the Z and N flags do with a load of $FF? Or $31? Find out, and meet me at the start of the next chapter.

8.
Branches: Decision Making

If you're like me, the first Basic program you ever saw didn't do much for you. It probably went something like this:

```
100 INPUT "WHAT IS YOUR NAME ";A$
110 PRINT "THAT'S A NICE NAME, ";A$
120 END
```

Unless you were exceptionally creative with your input, (THAT'S A NICE NAME, GRAND CAYMAN ISLAND) it wore thin quickly. But my first encounter with testing and looping was almost a religous experience.

```
100 N = 0
110 PRINT N , N * N
120 N = N + 1
130 IF N <= 10 THEN 110
140 END
```

Somehow the concept of testing and, if necessary, repeating a series of instructions was facinating: "Wow, I could change the 10 in line 130 to 1000. . . or 1000000 . . . Or change line 110 to print the cube root too!"

Put simply: **Decision making and looping are what computers are all about.** This is as true for machine language as it is for Basic. To execute our first decision-and-loop 6502 program we'll need some new instructions: The **Decrement / Increment** series, and a Branch or two.

There are 4 implied (one byte) instructions to increment (increase by one) and decrement (decrease by one) the contents of the X and Y registers. They are: DEX, DEY, INX, and INY. In table form:

MNEMONIC	OPCODE	OPERATION
DEX	$CA	Decrement X register
DEY	$88	Decrement Y register
INX	$E8	Increment X register
INY	$C8	Increment Y register

These instructions have a "wraparound" effect. If you decrement a register that contains $00, it goes to $FF. If you increment a register that contains $FF, it goes to $00. There is also a way to inc/dec memory locations. Strangely enough, there isn't an inc/dec pair for the accumulator, although there is a way to accomplish the same thing.

Like loads and transfers, these instructions condition the N and Z flags. If we execute DEX at a moment when the X register contains $01, we get $00 in X and a set Z flag. This makes the inc/dec instructions useful in counting loops. Load the X (or Y) register with the number of times you want the loop to occur. Next, do the operation(s) you intend to repeat. Now decrement X to reflect that you've been through the loop one time. Last comes something that can both test the Z bit, to see if X has been reduced to zero yet, and depending on the result of the test, cause us to jump back and repeat the process again.

These conditions are met by the **BNE** instruction. Pronounced "Branch if Not Equal", with an opcode of $D0, this instruction is the equivalent of the Basic statement:

IF A <> 0 THEN GOTO 1000

BNE is one member of the branch family of instructions on the 6502. There are seven others, two for each of the four testable flags of the P register (**C**, **N**, **Z**, and **V**). One that tests for the desired bit set, another for the same bit clear. In table form:

MNEMONIC	OPCODE	OPERATION
BCC	$90	Branch on carry clear
BCS	$B0	Branch on carry set
BEQ	$F0	Branch on result = 0 (Z Set)
BNE	$D0	Branch on result ≠ 0 (Z Clear)
BMI	$30	Branch on result minus (N Set)
BPL	$10	Branch on result plus (N Clear)
BVC	$50	Branch on overflow clear
BVS	$70	Branch on overflow set

Branches are said to use **relative** addressing because of the way they are executed. A branch instruction is two bytes long; an opcode byte (which tells the 6502 what bit to test, and for what value), and a second, **offset** byte to tell it where to go if the test passes. This "telling it where to go to" is tricky, and has to do with why their addressing form is called **relative.**

If the condition specified by the test is true, then the second byte is used to calculate a new value for the program counter. The program counter, remember, is the placemark in memory that keeps the 6502 executing instructions in sequence. If the test fails (as it would for a BNE when the Z bit is set), the program counter advances normally by one and things proceed as if there had been no branch instruction at all.

If the test succeeds, the program counter is modified by having the second byte **added to it.** For example, if PC contained $805 (having just read from memory the second byte, say a $10, of a BNE instruction), and the 6502 determines that the test has passed, it forms the new PC by adding $10 to the $805 already there. The next instruction to be executed would be the one at $815 ($805 + $810).

Does this mean that branches can only happen in the forward direction? No, negative branches are possible, although understanding how a negative branch is calculated is a little more difficult. If the data byte of the branch instruction is $80 or greater (Hint: bit 7, the sign bit, set), the 6502 knows to do a **subtraction** on the program counter rather than an addition. We will leave the details of this subtraction until a later section. (Sneak preview: $FF = -1, $FE = -2, $FD = -3...) Branches, then, can go either way, depending on the data byte, by making the program counter either larger or smaller. We may branch about 128 bytes in either direction.

Branching is demonstrated in PROG6. BLOAD and disassemble (L) it. It begins by loading X with $04; we are evidently intending to do something four times. Next are two set/clear instructions, there only to give the program some busy work to do in the loop. Next comes the new instruction DEX. DEX conditions the Z flag– if it didn't, this program wouldn't work. The branch instruction BNE consists of an opcode byte ($D0) at $804 and an **offset** ($FB) at $805. $FB, being greater than $80 has a bit 7 set, and therefore is a negative branch; the program counter will be reduced some amount if the BNE test passes.

```
0800:a2 04     ldx #$04
0802:38        sec
0803:18        clc
0804:ca        dex
0805:d0 fb     bne $0802
```

The TVC disassembler goes out of its way to help you understand where the branch will end up if the test passes. BNE $802 means "Branch if not equal to location $802". This is friendlier than saying just BNE $FC, and leaving you to figure out where the branch will go.

The first time we encounter the DEX instruction, X will be reduced to $03. This is non-zero, so Z will be cleared and the branch test will succeed, causing the loop to repeat. Finally, after 4 repetitions, the test fails and BRK ends the program.

Since this program is significantly longer in execution time (though not in length) than previous programs, now is a good time to learn some ways to control the speed of simulator execution. We've been using **Step 3** exclusively. What do the other step values do?

Step 2 executes an entire instruction without pausing at each micro-step. You can force the simulator to pause by pressing any key. When the instruction is over, you are returned to the monitor.

Step 1 is like **Step 2**, except that you don't enter the monitor between instructions, but instead plunge ahead with the next instruction. **Esc** forces the simulator to enter the monitor at the completion of the current instruction.

Step 0 is TVC's **high gear, flat out** speed mode. It saves time by skipping the process of writing to the screen during execution. The only things updated are the disassembly window and the next instruction area. The registers will not reflect their true values until you return to the monitor. "Flat out" and "high gear" are relative terms. In "high gear", TVC operates at something on the order of one **millionth** as fast as a real 6502.

If you are in step modes 1, 2, or 3, you can slow down or speed up the action by typing one of the number keys (1-9), while the simulator is running. 1 is fastest, 9 slowest. Now press return, and happy looping.

9.
Addressing Modes

We've moved so quickly that we've glossed over some very good questions you might have had. One being: "If there are only 56 instructions, why are there 151 opcodes?" The answer is tied up in something called **addressing modes.**

The 6502 is **good** at addressing modes; in fact, it makes some of its contemporaries (like the Z-80) look positively anemic in this regard. In a nutshell, **addressing modes determine not what instruction to perform, but where to get the data the instruction will use.** So far the demonstration programs have worked with a small subset of the many addressing modes available on the 6502. All loads have used **immediate** addressing, the form that tells the 6502 to load a register with the next byte following in memory. All stores have used **zero page** form, which specifies a memory location on page zero.

What if we wanted to load the accumulator, not with a number that we knew ahead of time when the program was written, but with the **contents of a memory location outside the program.** The Basic statement:

```
100 A = 14
```

is the equivalent of the way we've loaded the accumulator so far. More common in Basic is the statement:

```
100 A = B.
```

Accomplishing this in 6502 machine language requires a LDA of a different color. There is another opcode that decodes as LDA, but not the LDA-$A9 that makes the load occur from the next byte. It's LDA-$A5, and it makes the load occur from **the memory location specified in the next byte.** This is a slippery idea, I'll admit, but crucial to your future happiness as a world famous machine language programmer.

PROG7, another two-byte special, will clear up the mystery. Bload and list it. Notice that the disassembly is not quite identical to that for PROG1.

PROG1

```
0800:A9 33    LDA #$33
```

PROG7

```
0800:A5 33    LDA $33
```

Opcodes $A5 and $A9 cause the disassembler to produce the same mnemonic, LDA, but **different** operands. The "#" is your clue to understanding what kind of LDA you've got. By 6502 convention, a pound sign in the operand means that the value to load is "immediate", contained in the byte occurring next in memory. The **absence** of the pound sign in the second instruction tells us that the load will occur from the memory location specified in the operand, in this case from location $0033.

We just learned a new opcode, $A5, but not a new instruction. $A5 is LDA using the **zero page** addressing mode. $A9 is LDA using **immediate** addressing. Now execute PROG7. Pay close attention to how it gets $0033 into AD. Similar to the STA $33 instruction of PROG2.

What, there's more? Now a third way to LDA. Some of you have been asking: "What if I want to load the accumulator with a value stored somewhere in memory, but not a location down in page zero? Say an address like $A09 or $BFFF?"

Very good question. And yes, there is a way to do it. You may specify any of the 65,536 locations using **absolute addressing.** An instruction using absolute addressing requires three bytes: An opcode byte, and two bytes that specify the memory location the operation is to use.

Bload PROG8 and list it. PROG8 will load the accumulator from $B1C, when we let it, which we will in just a second. First, a close examination of the disassembly.

```
0800:AD 1C 0B LDA $0B1C
```

Notice that the **least significant byte of the address comes first.** 6502 convention is to store two byte values in sequential memory locations with the least significant byte stored first (lowest address). There's no special reason for this; they just adopted a convention and stuck with it. Again we find the disassembler working hard to make life easier for us. It rearranges the operand into normal left-to-right form. It's a lot easier to grasp "LDA $0B1C" than "AD

1C 0B".

Now execute PROG8. As you might expect, it takes longer to run than the other forms of LDA we've used. The data buffer is used to temporarily store the first byte of the address until we're ready for it. However, except for the extra memory fetch and transfer to the address bus, it runs exactly like the other two, finishing up with a flag conditioning and a final increment of the program counter.

So there you have it. One instruction, LDA, and three different opcodes ($A9 for immediate; $A5 for zero page; $AD for absolute). Can we use absolute addressing to access zero page locations? Yes, you may. There is no rule against the instruction:

0800:AD 12 00 LDA $0012.

If we can do that, why is there a zero page addressing mode at all? **Because only two bytes are needed instead of three.** Absolute addressing takes more storage and more time to execute. For efficiency, 6502 programmers place their most frequently accessed variables in page zero. As a result, the zero page is prime real estate in the 6502 memory map. Although in theory you can use page zero for program storage, this is rarely done. It would be like using a square block in downtown Chicago to grow tomatoes.

There are only 256 locations, and everybody wants to use them. If you're writing a machine language program that will be called from Basic, you'll have to be careful to use zero page locations that Applesoft, DOS, and the Apple monitor routines **don't** use. To determine what locations are safe, consult the chart on pages 74 and 75 of the **Apple II Reference Manual.** (If you don't have a copy of the **ARM** already, get one. It is a jewel, chock full of facts you'll be needing to write 6502 programs. A tribute to Apple's philosophy of letting people know as much as possible about Apple machines, so that they can write programs and build hardware to make more people want to buy them. This sounds perfectly logical, but it was a breakthrough in the traditionally secretive computer industry.)

The load and store instructions of the index registers have these addressing modes also. This table summarizes the opcodes for all three addressing modes for LDA, STA, LDX, STX, LDY, and STY.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
LDA	$AD	$A9	$A5	Load accumulator
STA	$8D		$85	Store accumulator
LDX	$AE	$A2	$A6	Load X register
STX	$8E		$86	Store X register
LDY	$AC	$A0	$A4	Load Y register
STY	$8C		$84	Store Y register

There are no opcodes for stores in immediate addressing mode. But then, what on earth would you do with an instruction that stores a register in an address location inside your program?

A final ominous word before we move on to more jumping around fun in the next chapter. I said earlier that the 6502 is a champion at addressing modes. You don't get to be a champion having just three modes for a popular instruction like LDA. You get to be a champion by having **eight.**

10.
Subroutines: The Stack

Next on the agenda are three instructions that, like a successful branch, alter program flow. Changing program flow means changing the program counter. Unlike the branches, the 6502 has no choice in the matter.

The new instructions are: **JuMP** (JMP, $4c), **Jump to SubRoutine** (JSR, $20) and **ReTurn from Subroutine** (RTS, $60). All three have direct counterparts in Basic.

MNEMONIC	OPCODE	OPERATION
JMP	$4C	Jump to new address (Basic GOTO)
JSR	$20	Jump to subroutine (Basic GOSUB)
RTS	$60	Return from subroutine (Basic RETURN)

JMP is a three byte, absolute instruction that puts the address of our choice in the program counter, thus shuffling us off to wherever in memory we've got instructions that need executing. As with all absolute instructions, the address we're jumping to is stored in memory with the least significant byte first. One use for JMP is to extend the range of a branch. A branch on its own is limited to about 128 bytes in either direction. If you use a branch in combination with a JMP, you can go as far as you want.

```
Instead of:  0810: BEQ $F000 (no can do)
                   ETC...
use:
             0810: BNE $0815
             0812: JMP $F000
             0815: ETC...
```

Bload PROG9 and list it. PROG9 is full of jumps–six of them, to be exact. But the disassembly just lists the first one. If you want the disassembler to show you what's out there waiting at $900 after the first jump, you have to ask for it specifically.

Now execute it. What you have at the end of PROG9 is an **infinite loop.** Like a cat chasing its tail, this program will never go anywhere. Although not a problem when we're executing programs with a simulator that lets us quit with a press of escape, it can be a serious problem under real 6502 execution. Infinite loops can only be broken by pressing reset. (The reset key is connected to the 6502 in a more intimate way than the rest of the keys, and has an impact on it more like the power switch than a keypress.)

A different sort of jump is controlled by the JSR/RTS pair. They're used like the GOSUB/RETURN combination of Basic. In fact, most every programming language has some way to implement this concept.

Executing a 3 byte (absolute) JSR instruction will, just like a JMP instruction, divert program flow to the address contained in the operand portion of the instruction. But with an important difference: Just before it goes to the new address, the 6502 **saves where it is now**, by storing the current contents of the program counter in memory. This enables the 6502 to find its way back when it finishes the subroutine.

How JSR and RTS work. Even though it is not strictly necessary to understand the underlying mechanics of the JSR/RTS pair to use them, I'm not going to let you off that easy. That's okay for Basic programmers, to accept a gift without worrying about where it came from. Machine language programmers look every gift horse square in the mouth to see the pitfalls lurking there.

JSR and RTS use something called the **stack** to accomplish the feat of returning after a subroutine has been completed. The 6502 stack is two things, working together: the stack pointer register (S), and $100 bytes of memory ranging from $100 - $1FF, the stack page. Although there is nothing to stop the machine language programmer from using the stack page of memory for general purpose program and data storage, it is **strongly** recommend that you reserve this area for the stack. With freedom comes responsibility.

The Classic Cafeteria Tray Analogy. The stack can be visualized as a stack of trays in a spring loaded container at the beginning of a cafeteria line. The tray at the top, ready to be pulled off next is the one most recently entered. The one at the bottom may have been there since Mother's Day. This is called a **LIFO** data structure, for Last In, First Out. As opposed to a grocery store line, which is **FIFO**, First In, First Out.

If we put two green trays on a stack of red ones, we know that the next two trays pulled off will be green. To implement the stack for useful purposes of storage, we need only two operations: Push (put a tray on the stack) and Pull (take a tray off the stack). We don't care if there are 50 trays or 15 when when we issue a Pull command. We only care that we get the one most recently put there. If I push a $45 (a $45 written with a Marks-a-Lot on a tray) onto the stack, and then an $FF, when I turn around and execute a pull, I'll get the $FF back first.

How is a one byte register and $100 memory locations like a cafeteria? The trays are one byte numbers that the 6502 will push and pull. The holder is the stack page—but instead of moving all 256 bytes down one everytime we push a value on the stack, the only thing that moves is the contents of the stack pointer. **The stack pointer always points to the most recent entry in the stack minus one.** If S contains $FC, and we execute a push, the value we push winds up stored at $1FC, and S is decremented to $FB. The first position in the stack is $1FF, and subsequent entries (i.e., more recent ones) use successively lower memory locations.

MICROSTEPS OF A PUSH

Transfer stack pointer to ADL. ADH = $01 (for stack operations, ADH is "hardwired" to 1 to force address references to be in the stack page)

Transfer register to be stored to data latch

Write

Decrement stack pointer

MICROSTEPS OF A PULL

Increment stack pointer

Transfer stack pointer to ADL. ADH = $01

Read

Transfer data latch to selected register

By convention, the stack pointer always points to the first vacant space in the stack. A Pull therefore increments the stack pointer **before** the read; a Push decrements the stack pointer **after** the write. Now that you're thoroughly confused, watch PROG10's JSR-RTS pairs put the stack through its paces.

Bload and list the first few instructions. As with programs that contain JMPs, the disassembly shows the first five instructions in sequence, not the code at the destination of a JSR. If you want to see that code, you'll have to ask for it.

This program "calls" (to use a popular synonym for gosub) a routine at $A00 to load the X and Y registers with $FF's, and a second routine at $900 that stores X and Y in a pair of consecutive zero page addresses.

The things to watch: **JSR**s put data on the stack (what data? The two halves of the program counter, PCH and PCL). **RTS**s pull data off the stack and into the program counter. For this program, put the window in memory mode, and use **RC 1F8** to display locations $1F8-$1FF. That's where the action will be. Since the window is in memory mode, the read microstep will be executed but not displayed.

Note that PCH is pushed **first** during JSR, and so must be pulled **last** during RTS. The address that goes into memory is the address of the last byte of the JSR instruction. RTS takes care of a final increment of PC to fully restore it to where we want to be, pointing to the instruction after the JSR. Also notice that pulling a byte from the stack does not erase it; it is not changed until something else is pushed there.

The stack for its own sake. There are four other instructions that use the stack. They are implied, one byte commands to push and pull the accumulator and P registers. In table form:

MNEMONIC	OPCODE	OPERATION
PHA	$48	Push accumulator on stack
PLA	$68	Pull accumulator from stack
PHP	$08	Push processor status register
PLP	$28	Pull processor status register

You will probably not have occasion to use PHP or PLP for awhile, even though this is the only way to load or store the P register. Usually P just sits there.

PHA and PLA are used to temporarily store a number without tying up a register or memory location. Suppose the accumulator contained the result of an important operation, but before we can use that result, we need the accumulator for another calculation. We have two options: Save the intermediate value in an unused register or memory location, or, push it on the stack. In many cases the latter course is best. When we are ready for the intermediate value, we pull it back into the accumulator.

There are two things to watch out for when you use the stack for data storage: First, there are a limited number of bytes in the stack and you will overwrite data with the 257th push (wraparound effect). If you are sharing the stack with Applesoft and DOS (such as when a machine language program is called from Basic), you have even fewer stack bytes available.

Second, if you are currently "within" a subroutine (i.e., a JSR has been executed without a corresponding RTS), you must be careful not to tamper with the stack so that the RTS will not work. This can happen two ways: Pushing a number and not pulling it before the RTS, or pulling a number without a preceding push. Both cause RTS to use two bytes that point somewhere, but **not** to the end of the JSR that called this routine.

PROG11 demonstrates care and feeding of the stack. The first subroutine (at $900) is a painfully slow delay loop. How can we get out? (We're willing to accept on faith that eventually X will be reduced to zero, and RTS executed.) By getting out, I mean getting back to the main loop that called this subroutine. Pretend you don't remember that we started at $800.

There are a couple of ways to do this. We could haul off and use the monitor to load X with 1 (doesn't take long to decrease a 1 to zero), and let the RTS occur normally. Or, we could peek into the stack page, figure out what bytes are the return address of the subroutine, and load the program counter (plus one, of course) with those numbers.

The easiest way is the monitor **POP** command. Executing a POP places the top two bytes of the stack (plus one) in the program counter, and increments the stack pointer by two. POP is the monitor's equivalent of RTS, and is useful in situations where you weren't watching closely and got into a subroutine without knowing how you came to be there. POP the address of the calling program to find out.

The subroutine at $A00 demonstrates how **not** to use PHA and PLA. By the time we get to the RTS that should return us the main program, the data at the top of the stack is part return address, part left-over pushed accumulator contents. **Ouch.**

Jump Back. Both JMP and JSR are three byte instructions using absolute addressing mode. JMP has a second addressing mode called indirect, opcode $6C. In mnemonic form:

```
JMP ($0900)
```

Like JMP, absolute, JMP (IND) is a three byte instruction that diverts program flow, without saving a return address; the mechanism for determining the address jumped to is different, however.

JMP ($2000) tells the 6502 to jump to **the address stored in memory locations $2000 and $2001.** Not to jump to $2000 and start executing code–but to look there for the values that will be placed in the program counter. If $2000 contains $F0 and $2001, $FD, then the program counter will end this instruction containing $FDF0. This is conceptually one level deeper than a normal JMP and you are entitled to feel a bit queasy at this moment. If you think of JMP as a load instruction for the program counter (which it is; we just don't call it that), then JMP absolute is a load immediate. JMP indirect is a load absolute. Since the program counter is 16 bits wide, two loads must be made from sequential locations. With a little imagination, the operand's use of parenthesis implies how the indirect jump works.

A bug ("feature") of the 6502 is its failure to properly handle indirect jumps that cross page boundaries. JMP ($20FF) will fetch the bytes from $20FF and **$2000** to form the new program counter, instead of from $20FF and **$2100.** This quirk has been faithfully copied in TVC.

PROG12 contains an indirect JMP. The first time through, after the instruction: JMP ($0A00) we end up at $810. Later, this same instruction puts us somewhere else.

Now, a message from our sponsor. Why should machine language programmers organize their programs in subroutines? For the same two reasons that a smart Basic programmer does: For memory efficiency, so that separate parts of a program may share a section of code without each having to duplicate it. And, for clarity of structure.

If you are to become a successful machine language programmer, you will need to make things as easy on yourself as possible, by writing programs that are clear and easy to follow. The "rat's nest" technique of jumps to jumps to jumps will have you spending more time figuring out what you did yesterday than on today's work. A good structure for machine language and Basic programs is to use subroutines liberally, sometimes even if they are called only once.

The Ideal Basic Program

```
100 GOSUB 1000
110 GOSUB 2000
120 GOSUB 3000
130 GOSUB 4000
140 GOTO 100
```

The Ideal Machine Language Program

```
LOOP:  JSR $1000
       JSR $2000
       JSR $3000
       JSR $4000
       JMP LOOP
```

To climb down from my soapbox, let me say in closing that even in well structured programs, you will make enough mistakes to satisfy your inborn programmer's desire for debugging sessions.

11.
Instructions That Work: ADC/SBC

So far we've haven't learned any instructions that really sink their teeth into a programming problem. We've loaded and stored and jumped over, under, around, and through, but haven't accomplished much in the process. A 6502 with only the instructions we've learned so far would be like a car with a great stereo, and plush seats, but no engine. This section introduces a pair of high octane computational instructions, ADC (add with carry) and SBC (subtract with borrow). These instructions may use any of the three all-purpose addressing modes we've used so far.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
ADC	$6D	$69	$65	Add with carry
SBC	$ED	$E9	$E5	Subtract with borrow

We've made reference to the accumulator's importance without saying **why** it's such a popular place; now we'll see. The accumulator is where numbers have to be to have SBC and ADC operations performed on them. You can't use any other register.

To add $23 to $14, load the accumulator with $23 and ADC $14 to it. The answer, $37, replaces the $23 that was in the accumulator. Results **accumulate** there. The accumulator is always involved in half of a computation and holds the result.

The operation of the ADC instruction is as simple as adding two eight bit numbers, something that humans learn to tackle in the second grade. The only thing remotely tricky has to do with why it's called "ADC", add with carry, and not just "ADD". The word "carry" means exactly the same process that humans use when they add numbers on paper.

```
   1          11
   34          66
+  19       +  44
   53         110
```

The 6502 needs the carry flag to keep track of when an addition produces a result greater than can be held in the accumulator. The accumulator can't grow, so C is drafted to be its ninth bit. Is nine bits enough to represent the largest possible result of eight bit addition? Check it out.

```
  $FF
+ $FF
 $1FE  (1 1111 1110)
```

Apparently so. Anytime an ADC produces a value greater than 255, the carry flag is set.

```
  $7F                      $31
+ $82                    + $16
  $01  + a carry           $47  no carry
```

ADC also conditions the Z and N flags, according to the same rules we have already learned for these flags. If an ADC causes a zero to be in the accumulator, the Z bit will be set. If it causes bit 7 of the accumulator to be set, then the N flag will be set. Otherwise, N and Z will be reset.

Not only does ADC condition carry going out, it includes carry in the addition; if carry is set going into an ADC, as the result of a SEC instruction or a previous ADC, the result will be **one greater** than otherwise. This is a slight annoyance when we need to quickly add a couple of eight bit numbers, as we must execute a CLC before ADC to insure that we get the right answer, but is a blessing for more complex calculations, as we shall see.

PROG13 demonstrates ADC in action, using immediate addressing. Bring in PROG13 and execute it. Play around with different values for the data bytes until you are comfortable with your understanding of how ADC computes a new value for A, based on the operands and the carry bit going in, and second, its conditioning of the Z, N, and C flags going out.

Despite the potential for confusion in having to consider the state of the carry bit on every addition, the C flag is more boon than bane, since most uses human beings have for the 6502 involve numbers greater than 255–and the carry bit is crucial to working with larger numbers.

Even though the accumulator is limited to 8 bits, it is possible to add and subtract numbers much larger than 255 using **multi-precision arithmetic.** This means using 2 or more bytes in memory to represent values. How big a number can you store in two bytes?

$$2^{16} - 1 = 65{,}535$$

In three?

$$2^{24} - 1 = 16{,}777{,}215$$

We quickly come up to a range of useful magnitudes. PROG14 is a two byte addition, using the zero page forms of ADC, LDA, and STA. Before we run it, we'll need to EDIT some numbers into page zero for it to use. Do this addition:

```
  $13FC  (A)        5116
+ $4597  (B)      + 17815
  $????  (C)      -------
                    22931
```

You might want to first run this problem through the calculator to see if the program produces the same result (it better!). We're going to use zero page memory locations $00 - $05 to store operands A and B, and the answer, C. Use $00 and $01 for A, $02 and $03 for B. Initialize $04 and $05 with zeros. Use EDIT mode to write the data into memory. As always, LSB first (in lowest location). $00 should get $FC and so on.

As is our wont, run the program a few times with different data. What happens if your addition produces a value greater than we can store in 16 bits? Is the carry flag still enough to handle the result?

SUBTRACTION

The 6502 also has an instruction for subtracting one byte numbers, **SBC**, Subtract with Borrow. It functions more or less like ADC with a confusing twist. Like ADC, it uses the accumulator for the first operand and a selected memory location for the second, with the accumulator getting the result. Subtracting 2 from $14:

```
LDA #$14
SBC #$02
```

The confusing part concerns the borrow flag; namely, **there is no borrow flag.** (B is the break flag, and has nothing whatever to do with subtraction.) **Borrow is defined to be the opposite of carry.** If C is set, borrow is reset; if C is clear, borrow is set. Confusing? You bet it is.

Take the subtraction:

7 - 2.

To do it on the 6502, place 7 in the accumulator, and execute SBC #$02. As with ADC, the answer includes the carry flag in some way. If C is set when this instruction is executed, you'll get 5 in the accumulator for an answer, because **a set carry bit means no borrow.** If C was clear, then the answer will be 4, because a clear C bit means a borrow occured previously.

Like ADC, SBC conditions the carry flag going out, too. Whenever a bigger number is subtracted from a smaller one a borrow is generated **(carry is cleared).**

$14	$14	$14
- $22	- $12	- $14
- $0E	$02	$00
Borrow	No Borrow	No Borrow
(Carry clear)	(Carry set)	(carry set)

PROG15 contains some exercises that demonstrate SBC and its backwards use of the carry bit. Bload and list it now.

```
PROG15

SEC              (Clear borrow, by setting carry)
LDA #$07
SBC #$02
CLC              (Set borrow, by clearing carry)
LDA #$07
SBC #$02
LDA #$14
SBC #$22
```

Experiment with different values until you understand how carry affects SBC operations going in and how SBC conditions carry going out.

MULTIPRECISION SUBTRACTION

As with ADC, situations arise that require multiprecision subtraction. PROG16 demonstrates a 2 byte subtraction. BLOAD and list it. PROG16 will subtract the two byte number stored at $02, $03 from the two byte number stored at $00,$01, and put the answer in $04,$05. Use EDIT to set up this problem:

```
  $73A1        29601
- $46B1        18097
-------      -------
  2C F0        11504
```

Now execute it. The carry bit winds up set at the end of this program, meaning **no borrow resulted from the overall subtraction of these two numbers.** And this is what you'd expect, since $46B1 is smaller than $73A1. Tinker with the values until you are able to predict everytime the behavior of the imaginary borrow flag going into and coming out of SBC instructions.

MULTIPLICATION AND DIVISION

Regrettably, the 6502 has no built-in multiply or divide instruction. Some of the newer microprocessors (8086, 68000, Z-8000) do. But with a little programming we can use multiple applications of adds and subtracts to produce the same thing.

To multiply **n** times **m**, add **m** to itself **n** times. To divide **n** by **m**, count how many times **m** can be subtracted from **n**. This sounds involved, and for a human it's not recommended, but a speedy little rascal like the 6502 can do this a hundred times in the blink of a hummingbird's eye.

PROG17 is an eight bit multiply. The values stored in $00 and $01 are multiplied together, with the result going to $02 and $03. Verify for yourself that two bytes are sufficient storage to cover the greatest possible 8 bit multiply. Before you execute it, you must give it some numbers to use. For reasons of time, keep $01 fairly small, say less than $10.

PROG18 is an eight bit division. The number in $00 is divided by the number in $01. $02 gets the quotient and $03 the remainder. These two programs only scratch the surface of the subject of machine language multiplication and division algorithms, i.e., there are **better** ways to do it.

COMPARE–TEST MASTER

A powerful tool in test-and-loop situations is CMP, Compare Memory with Accumulator. There's also a CPX and a CPY for the index registers. A compare subtracts the selected memory location from the accumulator (or X or Y) and sets the N, Z, and C flags accordingly but **does not affect the value in the accumulator.** So what good is a subtraction that doesn't affect the accumulator? Plenty.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
CMP	$CD	$C9	$C5	Compare memory with accumulator
CPX	$EC	$E0	$E4	Compare memory with X register
CPY	$CC	$C0	$C4	Compare memory with Y register

Before we plow ahead with a program to demonstrate CMP, a digression. Most of you have seen or written Basic programs with the line X = PEEK (-16384), or something similar. Memory location -16384, aka $C000, is where the keyboard is hooked into the 6502's memory. If a Basic or machine language program looks at this location, it can discover what keys the human is pressing.

The basic idea is: Every key has a number associated with it. $0D is the return key. $1B is escape. $30 is "0". $32 is "2". Some keys (shift, control) have no value of their own–but change the code produced by other keys.

When a 6502 program fetches a number from $C000 it gets the number of the key most recently pressed. Apple uses the ASCII ("**ass**-key", American Standard Code for Information Interchange) character set used by most computers and peripherals, so it is fairly straightforward to hook an Apple up to someone else's machine (like an Epson printer) and have them agree on the number that represents a comma and so on. The ASCII character set is in appendix B, and you probably have 4 other copies around somewhere. (You can never have too many copies of the ASCII chart.)

There is a complication to the keyboard story. The 6502 is so fast, that if a normal human being depresses a key in a normal human way, holding it down for something on the order of a tenth of a second, the 6502 could read the keystroke, go and do something with it (like display it on the screen), and come back and **get the same keystroke again.** You might get 36 apostrophes instead of one.

In practice, this doesn't happen. If you press one key you get one character. The reason has to do with the seventh bit of location $C000. Bit 7 is called the keyboard strobe. The electronics in the keyboard set this bit whenever you press a key. It **stays** set until the 6502 specifically clears it. This is not done by writing a zero to $C000 (this isn't RAM, remember), but by addressing I/O location $C010, the keyboard strobe. The keyboard strobe is an address dependent switch; the act of accessing $C010, regardless if with a read or write clears bit 7 of $C000.

These two facilities give 6502 programs a way to tell a fresh keystroke from one that's laying around from before. A subroutine to grab keystrokes might look like this (in quasi-flowchart form):

Loop: Load accumulator from $C000
Is bit 7 set? If No, Go to Loop
Clear keyboard strobe.
Return.

Clearing the strobe makes sure that **next** time we call this routine we will not get the same character unless it was typed again.

There's a program on the disk that demonstrates use of the CMP instruction to handle data plucked from the keyboard by a routine like the one shown above.

Bload PROG19. Prog19 first calls the GETKEY routine we outlined above. It returns when a key has been pressed, with the value of the keypress in the accumulator. Next comes an immediate addressing compare to see if we got the escape key. If we do, the program ends. If we don't, we repeat the whole process.

TVC uses the same keyboard address, $C000, as PROG19 (only one per Apple), so if you want it to read your keypress, you'll have to press a key just as the read microstep occurs–otherwise, TVC will interpret it as a pause order, and clear the keyboard strobe before PROG19 ever sees it.

12.
Beyond Adding and Subtracting

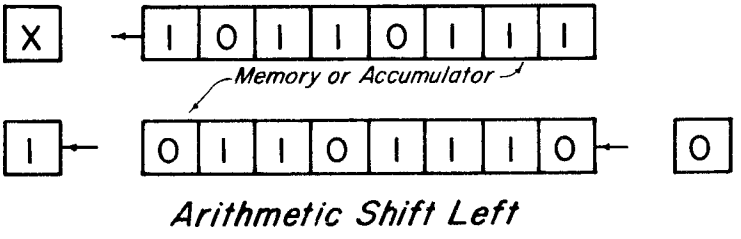
Thus far we've encountered two groups of 6502 instructions that actually get their hands dirty and perform calculations: The ADC/SBC pair, and the increment/decrement series. This chapter introduces two more groups of instructions to tackle problems with: The **logical** and **shift** instructions.

These commands differ from the ones seen previously in that they use the contents of registers (usually the accumulator) on a bit basis rather than on a cumulative basis. When we added $14 to $78 in the last chapter, we were happy to consider the $8C that turned up in the accumulator as just that: the quantity $8C. For the logical and shift instructions, however, we are usually more interested in the trees than the forest.

THE SHIFT INSTRUCTIONS

The 6502 has instructions for sliding all the bits in the accumulator one position to the left or right. As did ADC and SBC, these instructions use the carry bit as the ninth bit of the accumulator.

An ASL ("Arithmetic Shift Left") shifts all the bits in a memory location or the accumulator one position to the left. All the bits slide over one position to the left, bit 7 goes into C (whatever was in C is lost), and a zero replaces whatever moved out of bit 0. This chart demonstrates an ASL of the accumulator.



Arithmetic Shift Left

But what earthly good is it? A couple of things. First, it gives us a way to test any bit in the accumulator and branch accordingly. Suppose we've done an operation and we need to sample the contents of bit 5 and branch depending on what we find there. There is no **BA5**, "Branch on Accumulator Bit 5 Set", so we proceed as follows: Three consecutive ASL instructions to slide bit 5 into carry, then BCS to test and branch.

A shift left has the surprising effect of multiplying by two. Try it.

```
$20 (0010 0000)  X  2  = $40  (0100 0000)
$37 (0011 0111)  X  2  = $6E  (0110 1110)
$64 (0110 0100)  X  2  = $C8  (1100 1000)
```

You can multiply by four by doing two ASL's, by eight if you do three, and so on. Here are the shift instructions in table form:

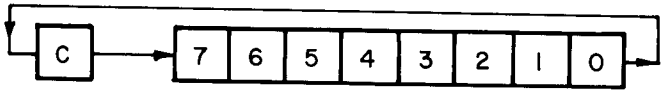
INSTRUCTION	ADDRESSING MODE ABS	ACC	ZP	OPERATION
ASL	**$0E**	**$0A**	**$06**	**Arithmetic shift left**
LSR	**$4E**	**$4A**	**$46**	**Logical shift right**
ROL	**$2E**	**$2A**	**$26**	**Rotate left**
ROR	**$6E**	**$6A**	**$66**	**Rotate right**

Shifts and Rolls of the accumulator are one byte, implied instructions, which for some reason are not classed with the other implied instructions, but rather are the only members of so called "accumulator" addressing.

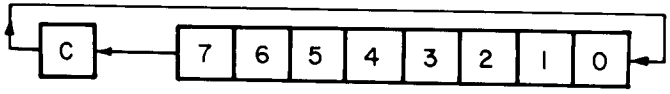
LSR is like ASL only we move right instead of left. Bit zero goes to the carry bit and a zero is shifted into bit 7. This **divides** the accumulator by two. Again, don't take my word for this. Experiment with the TVC calculator. The value left in the accumulator is the quotient; the values shifted out of bit 0 are the remainder.

The rotate instructions are slightly different. A rotate doesn't shift in a zero, it **rolls** in the contents of the carry flag.

ROTATE RIGHT



ROTATE LEFT



Rotations do not produce multiplication and division by multiples of two, unless you clear the carry bit ahead of time.

(Historical aside: ROR, the Hawaii of 6502 instructions, was the last instruction to be added to the 6502 instruction set. In fact, the earliest 6502's did not have ROR at all.)

None of the other registers may be shifted or rolled; however, you may shift and roll the contents of a memory location. Both absolute and zero page modes are available for shifts of memory.

PROG20 is a multiprecision shift. The two byte value at $900 and $901 (low order byte first, of course) is multiplied by four by the application of two ASL/ROL pairs. Shifting the low order byte puts its old bit 7 in carry; we get that value into bit zero of the high order byte by doing a roll of the high order byte.

THE LOGICAL INSTRUCTIONS

The standard assortment of logical operators are available to the 6502 programmer.

AND	Logical And.
ORA	Logical Or (Inclusive Or)
EOR	Logical Exclusive Or

Like SBC and ADC, these instructions operate on the contents of the the accumulator. In addition, they condition the Z and N flags according to the same rules.

Like the shifts, the logical instructions are cases where the trees are more important than the forest. What occurs in the instruction **AND #$33** is **eight simultaneous** logical ANDs of each bit of the accumulator and the corresponding bit of the selected memory location. For example:

```
      0011 0011 ($33)          1100 0000 ($C0)
AND   0100 0100 ($44)    AND   0100 1111 ($4F)
  =   0000 0000 ($00)      =   0100 0000 ($40)
```

One use for AND is to force selected bits of the accumulator to zero. To force bits 6 and 7 of the accumulator to zero, without affecting the other bits, AND the accumulator with $3F. To force every bit but 0 to zero, AND the accumulator with $01. Verify on paper that this works.

ORA is useful for **setting** selected bits. To set bits 4 through 7 of the accumulator, use ORA #$F0. To fill the accumulator with ones, use ORA $FF.

EOR can be used to complement a number (reverse the polarity of each bit). EOR #$FF will flip every bit in the accumulator; ones become zeros and zeros ones. This instruction is used in graphics programs in drawing a moving shape on a stationary background. By EORing twice, we can erase the shape without destroying the background. We'll actually do this later; for now, prove to yourself that two applications of EOR #$FF leave a number unchanged.

SPECIAL CASE: THE BIT INSTRUCTION

The last 6502 logical instruction is BIT, a peculiar hybrid of AND and CMP. BIT performs an AND operation between the accumulator and memory location–but, like CMP, conditions flags without altering the accumulator. As a bonus, BIT also transfers bit 6 and 7 of the memory location under test to the V and N flags, respectively. It is useful in checking I/O addresses that contain status information, particularly if bit 6 or 7 is the one that we're watching (as is the case with the keyboard strobe).

The logical instruction are supported by the three addressing modes we have encountered so far.

INSTRUCTION	ADDRESSING MODE			OPERATION
	ABS	IMM	ZP	
AND	$2D	$29	$25	And memory with accumulator
EOR	$4D	$49	$45	Eor memory with accumulator
ORA	$0D	$09	$06	Or memory with accumulator
BIT	$2C		$24	Test memory with accumulator

Sorry, no way to BIT immediate–but what would you do with that anyway?

PROG21 demonstrates AND and ORA setting and clearing bits in the accumulator. The subroutine at $A00 uses AND as a logical operator: If memory locations $900 and $901 both contain $FF, return with the accumulator equal to $FF. Otherwise, return with $00 in the accumulator.

13.
Indexing: Special Uses for X and Y

We mentioned in passing a while back that X and Y could be used as **index** registers. The time has come to find out what an index register is, and learn some new addressing modes in the process. So far we've encountered five addressing modes. Two of the five, **Relative** and **Implied**, are special cases; relative addressing is for branches only. Implied instructions (TAX, CLC) have no other form.

The other three addressing modes, **Immediate**, **Zero Page**, and **Absolute**, are more general, allowing the same instruction to be used in different situations. We have a choice in how we may load the accumulator; with a number contained in the instruction itself (immediate addressing), or with the contents of an address specified in the instruction (absolute and zero page addressing).

To this list of general purpose addressing modes we now add four **indexed** addressing modes: Absolute, X; Absolute, Y; Zero Page, X; and Zero Page, Y. Operands to indicate this addressing mode are as follows:

```
LDA $4000,X
LDA $4000,Y
LDA $00,X
LDX $00,Y
```

Indexing is best explained by presenting a problem that can't be easily handled by the addressing techniques we already know. Suppose we need to move a a cluster of $10 bytes residing in addresses $900 through $90F, to make room for something else. With the addressing modes we have learned so far, we can accomplish this "block move" with the following program:

```
LDA $900
STA $A00
LDA $901
STA $A01
LDA $902
STA $A02
LDA $903
STA $A03
etc...
```

And so on. To move all 16 bytes we'd need 32 instructions at three bytes apiece. Not very efficient to use 96 bytes of program storage to make room for 16 bytes of data storage. And what if we had to move 100 bytes? or 200? Wouldn't it be nice if there was a way to handle this situation with some incrementing and looping? Enter indexed addressing, in which the X and Y registers are used as **offsets** from a **base** address.

Bases? Offsets? Let me show you what I mean.

So far we only know one way to load the accumulator from $0903; LDA absolute. But what if we use a new addressing mode for LDA that provides a two byte **base address** of $0900, and tells the 6502 to **modify that base address with the current contents of the X register.** If we execute the instruction LDA $0900,X (hex form $BD $00 $09) at a moment when the X register contains three, the accumulator is loaded from $0903. If we then increment X and execute the same instruction, the accumulator will load from $0904.

Indexed addressing makes block moves a breeze. PROG22 demonstrates a more elegant solution to the move problem.

```
$0800 LDX #$00
$0802 LDA $900,X
      STA $A00,X
      INX
      CPX #$10
      BNE $802
      BRK
```

From 32 instructions, 96 bytes, to 6 instructions, 13 bytes. Quite a savings. And we can move as many as 256 bytes without the program growing one whit. As you step through this program, the thing to watch is the new microstep "CALC ADDRS" (calculate address), in which the address bus is modified by the X register. Otherwise, in conditioning of flags, and ultimate result, LDA absolute, X, is exactly like LDA absolute.

The same thing can be done with the Y register. This table summarizes opcode values for the load and store instructions for these new addressing modes.

	ABS,X	ABS, Y
LDA	$BD	$B9
STA	$9D	$99
LDX		$BE
STX		
LDY	$BC	
STY		

Notice that for the first time an opcode table has gaping holes. There isn't an opcode for LDX ABS,X. Nor is there one for LDY ABS,Y. NOT ALL ADDRESSING MODES ARE AVAILABLE FOR ALL INSTRUCTIONS. This is partially due to a logical conflict: Does it make sense to load the very register you've used to locate the memory location you're loading it with? But it stems mainly from the physical limitations of integrated circuit technology, circa 1975. Much as we'd like to have them, there wasn't room on the chip to provide every addressing mode for every instruction.

The most important instructions were given the most addressing modes: ADC, SBC, EOR, AND, ORA, CMP, LDA, and STA. Consult appendix F for the addressing modes available for each instruction.

INDEXING ON PAGE ZERO

That's two new addressing modes, Absolute, Y and Absolute, X. Indispensible, but like all three byte instructions, something of a memory hog at three bytes each. There are also two byte, space saving, Zero Page, X and Zero Page, Y addressing modes.

	ZP,X	ZP,Y
LDA	$B5	$B9
STA	$95	$99
LDX		$B6
STX		$96
LDY	$B4	
STY	$94	

PROG23 demonstrates zero page indexed addressing. Watch for wrap-around. If adding X to AD in the CALC ADDRSS phase produces a value greater than $00FF, ADL wraps around so that it always contains a zero page address. **LDA $80,X,** if executed at a moment when X contains $90, will load the accumulator from $10.

INDEXING, PART II

Indexing is a powerful technique that allows a looping program to repeatedly form different addresses with the same instruction. This section introduces two more indexed addressing modes: Indirect, Indexed and Indexed, Indirect.

Let's review the concept of **indirect** addressing. Way back in Chapter 10 we used indirect jumps. (Remember feeling queasy? That was JMP indirect). An indirect addressing mode doesn't specify the address to perform an instruction with–it specifies the address that **stores** the address with which to perform the instruction.

To review: An ordinary, garden variety JMP $B136 (absolute addressing) puts $B136 in the program counter and that's that. JMP **($B136)** instructs the 6502 to fetch the contents of locations $B136 and $B137 and use those contents to form the new program counter. This enables us to change where the jump points under program control.

The 6502 contains two addressing modes that use the indirect concept in tandem with the index registers. Two forms are available; one that uses the X register only, called indexed, indirect, and one that uses the Y register exclusively, called indirect, indexed. (Yes, the names **are** confusing.)

INDIRECT, INDEXED

Suppose you faced a situation that required a block move of greater than 256 bytes. You could tackle this problem with two consecutive applications of normal absolute, indexed addressing as shown below.

Moves $200 bytes from $3000 to $4000.

```
        LDX #0
LOOP1:  LDA $3000,X
        STA $4000,X
        INX
        BNE LOOP1
LOOP2:  LDA $3100,X
        STA $4100,X
        INX
        BNE LOOP2
        BRK
```

While this program would work, it is sorely lacking in elegance. Two loops instead of one. Tacky. If we needed to move four pages of memory we'd need four loops. Enter Indirect, Indexed adressing. In mnemonic form:

LDA ($45),Y

The operand's arrangement of the parentheses is a clue to how indirect, indexed addressing works. Since the Y is outside the parentheses, it's trying to tell us that the indirect portion of the instruction will be carried out **first**, and the indexing applied **second.**

For example, executing LDA ($45),Y: Memory location $0045 is read and the value stored in the data buffer. Next, location $0046 is read. Suppose we read a $00 from $0045, and a $20 from $46. We have now "indirectly" formed the address $2000, (as always, LSB first). Finally, apply indexing. If Y was equal to 6 when we executed this instruction, we will load the accumulator from location $2006. If we were to increment location $46 (making it $21), executing LDA ($45),Y again would fetch the byte stored at $2106.

Even though it takes several fetches of memory to execute an (IND),Y instruction, and consequently more time than other addressing modes, it is extremely efficient for code length. (IND),Y instructions require only two bytes; one to specify the instruction-addressing mode; the second, the first of the consecutive zero page addresses that will form the base address. Despite their two byte length, they can specify a location **anywhere** in memory. Indirect, indexed addressing is a big reason for the space crunch in page zero–every program needs a couple of zero page pointers. (Pairs of zero page locations used in this way are frequently called pointers because their contents "point" in memory to where an operation should occur.)

Only the Y register can be used this way. There is no LDA ($45),X instruction. There's a program on the disk named CLEARPROG that demonstrates (IND),Y addressing. Bload and list it. This program writes zeros to all $2000 addresses that make up the hires screen, effectively clearing it to black.

First it sets up a pointer pair ($FA and $FB) with the address of the first byte in hires page 1 ($2000, which happens to hold 7 dots in the upper left hand corner). Then it executes a loop, the active ingredient of which is the STA ($FA),Y instruction. After writing $00's to the first 256 locations, we increment the high order byte of the pointer, check to see if it's $40 yet (in which case we're done); and if not, repeat the process. Writing a $00 in a hires memory location produces a short black horizontal line seven dots long (the most significant bit holds color information), effectively erasing a small portion of the TVC display with each write.

Once you've watched a couple of cycles of this program under TVC execution, you probably think you've seen enough. If we let it run all the way through under simulation mode, eventually the screen would be cleared, but you would be bored into a coma. Now's the time to bring a couple of commands out of the closet.

MASTER MODE

Thus far we've been in non-master mode exclusively. This is a good place for beginners to be; non master-mode makes it just about impossible for you to hang up the computer (short of prying the 6502 out of its socket with a fingernail file). Writes to memory locations inside the TVC program are not allowed, certain I/O references that can do messy things are locked out. Most importantly, you are denied access to the GO command.

THE GO COMMAND

The GO command causes a program in memory to be executed not by the simulator but **by the 6502 itself.** If you are in master mode, and if the next instruction is a JSR, then TVC will pass execution of that subroutine directly to the 6502. When the 6502 encounters the RTS at the end of the subroutine, TVC will regain control and redisplay the X, Y, P, A, and PC registers with the values they acquired in the subroutine, and place the JSR in the disassembly window.

There are a million and one ways (conservative estimate) that a machine language program can go wrong, and almost all of them will cause you to lose control of the computer. You may spray a deadly hail of bytes into the TVC program–in which case you'll have to reboot, or, if you're lucky, your sick program will be stuck in a harmless infinite loop, and a reset should (no guarantees) restore control.

To GO CLEARPROG, first enter master mode with the command:

MASTER

The M flag on the status line illuminates. **You are now a Visible Computer Master.** (Feels great, I know.) A side effect of master mode is that you can no longer read the TVC disk (you can try, but you'll get I/O errors). However, you can now read and write to DOS 3.3 disks, something you'll be doing a lot more of down the road.

If you want to bload a program from the TVC disk, do it before you enter master mode, or, exit master mode with **MASTER OFF**, load the program, and reenter master mode.

Get the program counter pointing to the JSR instruction at $800. GO won't work if you're not on a JSR. Now **GO** it, and be prepared to not be bored. Doesn't take long, does it? RESTORE the display. Change the $00 at $80E to a different value and run the program again. $FF makes the screen all white. $55 makes nice pin stripes. Have fun while you can, because we're about to spin your head completely around.

INDEXED, INDIRECT

If you **liked** indirect indexed, you'll **love** indexed, indirect. Whereas indirect indexed addressing is only available with the Y register, indexed indirect is only available with the X register. Confusing? You know it. In mnemonic form:

```
LDA ($45,X)
```

Again, an examination of the operand and some educated guessing furnish clues to how this addressing form works. Indexed indirect uses X to index a **particular** pointer pair **out of many**, which then is used to point to an address in memory. By way of example:

Suppose the first eight bytes in memory held these values:

$00= $00	$04= $00
$01= $08	$05= $0A
$02= $10	$06= $00
$03= $09	$07= $0B

Eight locations make up four pointer pairs. The first points to $800, the second to $910, the third to $A00, and the fourth to $B00. Indexed indirect addressing uses the X register to **select one pointer pair of many.** LDA ($00,X) will load the accumulator from $800 if X is 0; from $910 if X is 2; from $A00 if X is 4; and from $B00 if X is 6. This addressing mode is usually used to select under program control which of several tables will be used in an operation. In practice it doesn't get as much use as (IND),Y, but the day will come when you'll be glad it's there.

Only the biggies of the 6502 instruction set have these modes available to them: ADC, AND, EOR, SBC, ORA, CMP, STA, and of course, LDA. For opcode values consult appendix F.

REVERSEPROG is a takeoff on CLEARPROG. It does an EOR #$FF on every byte in the hires display. You may remember that this has the effect of complementing every bit in a byte. PROGXX quickly (under GO execution, anyway) produces a negative, black-on-white version of the TVC display. At the very end it checks for a keypress; if it sees one, the program ends. If not, it goes back and EOR's everything again; this puts the display back to normal. Run REVERSEPROG a couple of cycles under the simulator. Ready to GO it? Sorry, you can't. There's no JSR at the start of the program.

You'll have to write a JSR instruction that calls REVERSEPROG at $800. Edit the instruction:

```
0300: 20 00 08 JSR $0800
```

into page three. Now set PC to $300, and GO.

When you hit a key to end the program you have a 50-50 chance of leaving the display in negative form (doesn't hurt anything; RESTORE if it bothers you).

14.
Some Fine Points

You may have noticed that this manual is filled with phrases like "this is a powerful group of instructions", or "this instruction gets a lot of use". This chapter concerns a couple that aren't so powerful, or don't get a lot of use, or both.

NOP (No OPeration), opcode $EA, implied addressing, doesn't do a thing. **Nada.** If you execute a NOP, the only effect is that the program counter will end up one bigger and a little time will have been wasted. What good is an instruction that does nothing? It has two uses: As a short delay in a carefully timed counting loop, and most importantly, as a means of plugging blank spaces in memory, usually as a debugging technique.

If you were debugging this program:

```
0800:20 00 10   JSR $1000
0803:20 00 20   JSR $2000
0806:20 00 30   JSR $3000
```

and determined that the second subroutine had a problem, you could quickly check the functioning of the third subroutine by writing over the middle JSR instruction with three NOP instructions.

```
0800:20 00 10   JSR $1000
0803:EA         NOP
0804:EA         NOP
0805:EA         NOP
0806:20 00 30   JSR $3000
```

When we execute this program now, after the subroutine at $1000 returns we fall through to the subroutine at $3000.

One word of caution: On pages 127-128 of the **ARM** there is a misleading chart that implies that any of the 105 undefined opcodes can be used as a NOP. This is not the case. Only $EA is NOP. If you put one of these undefined values in the instruction stream, TVC knows to not execute it, but a 6502 doesn't. It will do something undefined, i.e., **who knows what.** Appropriately enough, there is no demonstration program for NOP.

NOP's potential as an innocuous time waster brings up the subject of instruction execution times. Normally, we are only concerned that a program run **fast**, or at least fast enough. Sometimes, for example in tone generation routines, we have to know **exactly** how long an instruction takes to run. The basic unit of time for the 6502 is the **instruction cycle.** In an Apple II, one instruction cycle takes 1.023 microseconds (.00000123 seconds). All instructions require two or more instruction cycles to complete. In general, the less reading and writing of memory an instruction requires, the faster it runs. DEX and SEC are fast, requiring only 2 cycles. LDA $45 takes 3 cycles. LDA ($01),X requires six cycles.

A RARE ONE

RTI (return from interrupt) is a rare instruction in Apple programs, because under normal circumstances, interrupts never occur to return **from.** But first, the $64 question: What's an interrupt?

Three of the 6502's 40 pins are **interrupt** lines, places where circuitry external to the 6502 can impact its normal fetch/execute /fetch/execute pattern. The three lines are called Reset, Non Maskable Interrupt (NMI), and Interrupt Request (IRQ). All three cause the 6502 to stop what it's doing (executing some program or other) and do something else. Some are more courteous to the program that's being executed than others, however.

RESET

There are two ways to generate a RESET signal on the reset pin of the 6502: Turn the machine on, or press the reset key. Either method causes the 6502 to drop whatever it's doing and immediately do an indirect jump (or "vector") to $FFFC (i.e., to the address stored in locations $FFFC and $FFFD). For machines with the Autostart ROM (built since 1979) this program begins at $FA62.

The reset handling program (laid bare for all to see in the **ARM**) takes care of some busy work, like the familiar beep, making sure that the screen is properly set up to display characters, etc., and then pops the big question: **"Have I just been turned on, or was the reset key pressed?"**. It gets the answer by looking at a couple of bytes in memory. If they don't look just right, it assumes that it was just turned on, and goes through an array of startup, housecleaning jobs like clearing the screen, displaying "Apple II" at the top, and, if it finds a controller card plugged into one of the expansion slots, trying to boot.

If the two test bytes look okay, program flow is shunted to whatever activity is appropriate for continuing what was happening when RESET got pressed; If you were in Basic, you'll get back perfectly healthy Basic, with your program intact, but not running. There's no way it can pick up exactly where it left off, because we didn't save the program counter when the reset occured. The reset cycle is discussed in detail on pages 36-38 of the **ARM.**

INTERRUPT REQUEST

If we put a signal on the 6502's IRQ pin, we command the 6502 to drop what it's doing and do something else, but to **first save where it is now so that we can get back later.** This is done by saving the program counter and the P register on the stack. Once saved, we "vector" to the code pointed at by locations $FFFE and $FFFF, the highest two locations in the memory map.

Two bits of the P register are involved with interrupts. The I bit, interrupt disable, is used to "mask out" interrupts. If I is set, interrupts are disabled–the 6502 ignores whatever is messing around with the IRQ pin. You'd want to mask out other interrupts, for example, when you're in the midst of handling an interrupt request already.

BRK: THE WHOLE TRUTH AND NOTHING BUT

Have you wondered why the P register has a B flag that doesn't have anything to do with borrow? Or why BRK is sometimes called a **software interrupt**?

BRK causes the 6502 to behave exactly as though an interrupt request had occured on the IRQ pin. An indirect JMP is made to the same program, pointed to by $FFFE and $FFFF. How can the interrupt handling program determine if it got there because of a hardware break or a software break? By checking the B flag. If set, the interrupt was due to BRK. If reset, it was a bonafide hardware interrupt. Let TVC execute a BRK instruction in master mode, and you'll see the code that makes this decision.

If the interrupt was due to the IRQ pin (which is unlikely, since normal Apple hardware will never generate one) the RTI instruction is used to get back to where you were just before the interrupt happened, by pulling the program counter and P register from the stack.

In non-master mode, the simulator doesn't execute a BRK the way a 6502 does; It does what we ultimately use BRK for–as a signal to stop execution for debugging purposes. In master mode the simulator exe-

cutes BRK the **real** way.

If the 6502 encouters a BRK while executing a subroutine via the GO command, TVC will regain control and update the programmer's registers with the values they held at the moment of the BRK. By convention, the program counter is the address of the BRK instruction plus two. The message "BREAK" appears on the error line.

The purpose of BRK is not to assist in the execution of a useful 6502 program, but for debugging. By setting BRK instructions at key points in your program you can usually find out what's working and what's not.

NON MASKABLE INTERRUPT

Then there's the non-maskable interrupt. NMI is similar to IRQ, only it has a different vector ($FFFA) and it **may not be ignored.** One use for NMI (but not in the Apple) is to connect it to a power supply sensor. When the sensor gives warning that the incoming AC line has dropped below some minimum value, the time remaining to the system is short, maybe only a hundreth of a second or so. We can't afford to be polite and wait for another interrupt to finish. The NMI vector points to an orderly shutdown procedure.

SIGNED NUMBERS

All the programs we've seen so far have assumed that the numbers being added, subtracted, decremented, etc. were always positive. Many times machine language programs face the same problem as the overdrawn checkbook: How to handle numbers less than zero. Or, put another way, what shows up in the accumulator when we subtract 6 from 3? You won't see any minus signs anywhere, that's for sure.

Any guesses as to how to represent negative numbers? (It has **something** to do with bit 7, hint, hint.) As a suggestion, how about using the lower 7 bits as the absolute value of a byte, and the 7th bit as a sign flag. Thus:

```
0011 1111 = + $3F
1011 1111 = - $3F
```

and

```
0100 1010 = + $4A
1100 1010 = - $4A
```

That wasn't so bad, was it? Almost the way people do it–if there's no minus sign, numbers are positive; here, if we have a clear sign bit, we mean positive. A nice, sensible solution.

Using the most significant bit as a plus/minus indicator limits the range of values that can be represented with a single byte to 127 to +127. (Again the formula: $2^7 - 1 = 127$.) Two byte numbers can use the 7th bit of the most significant byte for the sign, with the remaining 15 bits storing the absolute value. This limits us to the range -32,767 through 32,767 (ring a bell somewhere about the storage limitations of Applesoft integer variables?).

But hold on. Even though this scheme has a certain pleasing logic, it has a non-trivial problem: **It doesn't work.** Adding 3 + -6 should produce -3. Does it?

```
  0000 0011  (3)
+ 1000 0110  (-6)
  1000 1001  (-9)
```

No. Any way you slice it, -9 is not -3. How about 26 + (-14)?

```
  0001 1010  (1A)
+ 1000 1110  (-0E)
  1010 1010  (-40)
```

Not even close. And there's another problem. We have two bit patterns that mean zero: "Positive zero", 0000 0000, and "negative zero", 1000 0000. Ouch.

Logical, maybe–correct, uh-uh. Rather than subject you to a whole series of potential solutions that don't work, let us proceed immediately to a way to represent negative numbers that **does** work, two's complement.

As with non-functional method #1, bit 7 still indicates whether a number is negative or positive. It's the other 7 digits that are handled differently. A two's complement is formed by complementing (reversing) each bit and adding one to the result. We represent -$19 with the **two's complement** form of positive $19.

```
 $19 = 0001 1001
-$19 = 1110 0110 + 1 = 1110 0111

 $64 = 0110 1000
-$64 = 1001 0111 + 1 = 1001 1000
```

While somewhat less logical than the first method, two's complement representation possesses the desireable property of actually working when we put it into action adding numbers. It also solves the problem of two zeros. There's just one, 0000 0000.

To perform the addition 3 + (-6), first express -6 into two's complement form:

-6 = two's complement of 6 = two's complement of 0000 0110 =

1111 1001 + 1 = 1111 1010.

Now do the addition:

```
   0000 0011  (3)
+  1111 1010  (-6)
   1111 1101  (?)
```

Since the result has bit 7 set, we know the answer is negative, and by performing a two's complement to switch it to positive, we can see if we got the right answer.

Two's complement of 1111 1101 = 0000 0010 + 1 = 0000 0011 = 3.

It worked. We got minus three for an answer. Since the function of this book is to get you started in machine language, not to win you the George Boole Chair of Binary Studies at Stanford, there will be no rigorous proof attempted here of why this method works. (Audible sigh of disappointment.)

To practice, use PROGXX to add one byte negative numbers to positive numbers. Represent negative numbers with two's complement form; if you're lazy (and/or smart), you'll use the calculator for this. Subtract the number you want in two's complement form from zero. (e.g., to obtain the two's complement form of $3411, perform the subtraction $0 - $3411.)

A final note: For many, signed arithmetic proves to be one of the most elusive aspects of machine language. If this presentation left you more confused than enlightened, take comfort in the fact that most machine language programs don't need signed numbers. And when the day comes, six months or five years from now when you'll need to know it, I think you'll find you can pick it up.

BINARY CODED DECIMAL

The Decimal flag (D) of the P register hasn't seen a lot of action. In fact, except for a couple of sets and clears back in Chapter 7, we've ignored it entirely.

The D flag controls how the SBC and ADC instructions work. If reset, as it has been so far in all the demonstration programs (or should have been) SBC and ADC perform standard binary arithmetic. If D is set, the 6502 adds and subtracts using **Binary-Coded Decimal** (BCD) numbers. BCD is a numbering system in a limbo somewhere between binary and decimal. Because you will almost certainly have no immediate use for additions and subtractions of binary coded decimal numbers (unless you're planning to write a Pascal compiler and are worried about rounding errors), no further mention of it will be made here. Except, **keep this flag clear or all your adds and subtracts will be wrong.** As a matter of fact, the very first instruction executed when an Apple comes to life during a power-on reset is CLD. Leave it that way.

15.
Putting It All Together

So far we've been using 6502 programs without considering how they were produced in the first place–we said BLOAD and there they were. Since the purpose of this manual is to get you **writing** machine language programs, it's about time we wrote one, taking an idea all the way to a working 6502 program.

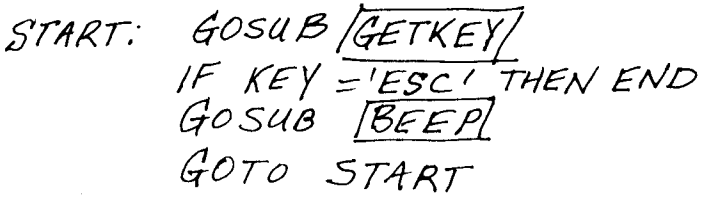
Hmmmm. What can we use for an idea... No, already done that. No, too complicated. How about. . . no, too easy. I've got it: **Play music with the Apple keyboard. Catchy name: "ASCII Organ".**

Now to flesh it out a little.

The higher the ASCII value of a keypress, the lower the note. We won't control duration (each note will last the same length of time). Escape exits the program.

The next step is to put the problem in computer terms, using a representation about halfway between the design's English and the mnemonics of 6502 language. If Basic seems a natural way to express the problem, feel free to use it.

ASCII ORGAN FLOWCHART



Next, translate this semi-program into the mnemonics of the 6502 instruction set. We're going to use **labels** in some places because we don't want to tie ourselves down to real addresses yet.

```
START:  JSR  GETKEY
        CMP  #$9B
        BNE  SKIP
        RTS
SKIP    JSR  BEEP
        JMP  START
GETKEY: LDA  $C000
        BPL  GETKEY
        BIT  $C010
        RTS
BEEP:   LDY  #$80
BEEP1   TAX
BEEP2   DEX
        BNE  BEEP2
        BIT  $C030
        DEY
        BNE  BEEP1
        RTS
```

This form of the program is called **assembly language.** It's not machine language yet–the 6502 can't cope with "JSR BEEP", anymore than it can understand the whispered command, "Beep the speaker, please". Before ASCII Organ can be run, we must "assemble" it into machine language.

```
800-20          START:  JSR  GETKEY
803-C9 9B               CMP  #$9B
805-D0 01               BNE  SKIP
807-60                  RTS
808-20          SKIP    JSR  BEEP
80B-4C 00 08            JMP  START
80E-AD 00 C0    GETKEY: LDA  $C000
811-10 FB               BPL  GETKEY
813-2C 10 C0            BIT  $C010
816-60                  RTS
817-A0 80       BEEP:   LDY  #$80
819-AA          BEEP1   TAX
81A-CA          BEEP2   DEX
81B-D0 FD               BNE  BEEP2
81D-2C 30 C0            BIT  $C030
820-88                  DEY
821-D0 F6               BNE  BEEP1
823-60                  RTS
```

Assembling each instruction into object code is a tedious, error prone job. After a half hour of flipping pages in reference manuals, calculating relative branch values, replacing labels with addresses, all of a sudden learning machine language doesn't seem like such a good idea after all.

Once we've translated the source program into a flock of bytes, (call it the **object** program), we must EDIT it into the computer. And hope we get every byte right, and don't change **JSR**s into **RTI**s along the way.

A test to see how good a job we've done of assembling and entering is to disassemble memory where we've placed our bytes and see if it resembles the original source program. It probably won't and we'll need to make a patch or two or three. Even once we **do** get it entered right, the program still won't work if the original source program had logical errors. If we do much rearranging at all of the source program, we'll have to re-assemble from scratch.

Yes, Virginia, there is a better way. The phase of the machine language programming process least suited to the talents of humans (and best suited to those of a computer) is the assembly itself. Wouldn't it be nice if we had a program that could assemble a source program **automatically**?

Happily, such programs, called **assemblers**, exist. An assembler converts 6502 assembly language ("source") programs into 6502 machine language ("object") programs. One fly in the ointment is that you won't have much luck getting an assembler to make sense of a pen and paper source program. You'll need a special program called an **editor**, a programmer's word processor, to produce the source program. Using an editor is ultimately faster than writing on paper, although it takes some getting used to.

```
1000 * ASCII ORGAN
1010 *
1020 START   JSR GETKEY
1030         CMP #$9B
1040         BNE SKIP
1050         RTS
1060 SKIP    JSR BEEP
1070         JMP START
1080 *
1090 *
1100 GETKEY  LDA $C000
1110         BPL GETKEY
1120         BIT $C010
1130         RTS
1140 *
1150 *
1160 BEEP    LDY #$80
1170 BEEP1   TAX
1180 BEEP2   DEX
1190         BNE BEEP2
1200         BIT $C030
1210         DEY
1220         BNE BEEP1
1230         RTS
```

This editor-produced source program for ASCII Organ looks remarkably like the hand written version, with a few exceptions. Every line is numbered. The editor that produced it uses line numbers as a means of editing in the same way Applesoft does.

Asterisks are this assembler's equivalent of Basic's REM. All lines beginning with an asterisk are comments intended to enlighten the person reading the source program. The assembler ignores them. The asterisk can be omitted if the comment begins to the right of an instruction in an area reserved for comments.

Once the source program is ready, in a separate step we command the assembler to assemble it into object code. A short program like ASCII Organ takes **four seconds** to assemble, with guaranteed accuracy. Is that better than half an hour, and making mistakes to boot? (Rhetorical question.) Once assembled, we can save the object program to disk, or run it, or whatever.

```
                    1000 * ASCII ORGAN
                    1010 *
0800- 20 0E 08      1020 START  JSR GETKEY
0803- C9 9B         1030        CMP #$9B
0805- D0 01         1040        BNE SKIP
0807- 60            1050        RTS
0808- 20 17 08      1060 SKIP   JSR BEEP
080B- 4C 00 08      1070        JMP START
                    1080 *
                    1090 *
080E- AD 00 C0      1100 GETKEY LDA $C000
0811- 10 FB         1110        BPL GETKEY
0813- 2C 10 C0      1120        BIT $C010
0816- 60            1130        RTS
                    1140 *
                    1150 *
0817- A0 80         1160 BEEP   LDY #$80
0819- AA            1170 BEEP1  TAX
081A- CA            1180 BEEP2  DEX
081B- D0 FD         1190        BNE BEEP2
081D- 2C 30 C0      1200        BIT $C030
0820- 88            1210        DEY
0821- D0 F6         1220        BNE BEEP1
0823- 60            1230        RTS
```

That's how the ASCII Organ came into existence. Now, how does it work?

ROUTINE BY ROUTINE

GETKEY is the same keystroke grabbing routine from Chapter 11. Waste not, want not. It returns with the ASCII value of the key pressed in the accumulator, bit 7 set.

When the main loop gets back control, it checks to see if the key is escape. If it was, we RTS to end the program. If it wasn't, call the BEEP subroutine, which produces a beep related in some way to the value stored in the accumulator. After the beep we jump to the top of the loop, and repeat the process.

BEEP deserves more coverage than "produces a beep related to the number in the accumulator". First, Y is loaded with $80. This is an outer loop counter that determines how many times we will repeat an inner loop, and ultimately the **duration** of the tone. The pitch is due to a delay loop based on A. We transfer A to X, and use X as the counter in an inner delay loop. After this delay, we do a speaker click; decrement Y (our duration counter), and if non-zero, reload inner loop (pitch) counter X, and repeat. We end up clicking the speaker 128 times, with a pause of variable length in between.

One more time, with test data. JSR to GETKEY and wait. After hundreds or thousands or millions of microseconds, **finally**, the human presses the X key. This causes the keyboard location to contain $D8 (check those ASCII tables), and makes the BPL test fall though to the keyboard strobe clear. Now $C000 contains $58, ($D8 with the high bit off). The accumulator still has the original, high bit set version. Which is promptly tested after we RTS to see if it's the code for escape, $9B. It isn't, so we take a detour around the RTS that would end the program, and JSR to BEEP.

BEEP always uses $80 for its outer (duration) loop. The $D8 from GETKEY figures into the **pitch** delay. Since the inner loop works down (DEX), the largest values for the accumulator produce the lowest tones, because longer delays between speaker clicks produce lower frequencies. Anyway, once we've run the inner-loop-toggle speaker combination $80 times, BEEP returns to the main loop, where the whole process repeats. A flaw of this program (I didn't say it was perfect, only that it has a good name) is that higher pitched notes have

shorter durations. Since the inner loop takes less time with smaller pitch-value keypresses, the beep routine as a whole takes less time.

PLAYING ASCII ORGAN

Make a couple of passes through ORGANPROG with the simulator. As with all programs that do keyboard checking, your keystroke will not be captured unless you press it during the read of $C000. You will find that even with great patience, and many passes through the loops of BEEP, that nothing resembling music ever occurs. The simulator is too slow to produce a tone. We need to click the speaker hundreds, or even thousands of times a second to do that. GO just the beep subroutine. That's more like it, beep-wise. Now GO the whole program. The first person to send in a tape of The Moonlight Sonata played on the ASCII Organ wins a special No Prize and a hearty "Well done".

BUBBLE SORT

A problem you will eventually face, probably sooner than later, is sorting. A common sorting technique is the bubble sort. There are more sophisticated sorts around, in fact, there aren't many **less** sophisticated, but when you're using machine language, and moderate amounts of data, there's no reason to get fancy.

A bubble sort works by "floating" the lightest (smallest) numbers in an unsorted list to the top. We start at the bottom and compare the bottom element with the next-to-the-bottom element. If they're not in the right order already, swap them and move up to the next pair. When we've been all the way through the list, the lowest number in the list is at the top. Has to be. Next, we repeat the entire process, except that we don't check the topmost number; we know it's the lowest already. After pass 2, the top two elements in the list are correct.

After as many progressively shorter passes as there are elements in the list, we are done. Let's work through a sample bubble sort on paper. The arrow points to the lowest member of the pair under test.

Starting at the bottom: Compare 22 to 77. Since 22 is less than 77, bubble it up a notch by swapping it with 77. Advance the pointer. Now the list looks like this:

Compare 22 to 19. We don't need a swap this time. Move the pointer.

Compare 19 to 34. Swap. Since the pointer is as high as it can go, this completes one full pass through the list. The smallest item in the list is now at the top. Move the pointer down to the bottom and repeat the process, only this time, we can stop one comparison sooner, since we know the top value is already correct.

When we've made three passes through the list, we're done.

The reason Basic sort programs are so slow is that even for short lists there's a lot of comparing and swapping to do. In the neighborhood of 32,000 comparisons for a list of 256 numbers, and roughly half that many swaps, depending on how well the list is sorted already.

Bload SORTPROG. It comes complete with $100 scrambled bytes in $A00 - $AFF. (Actually, the "scrambled bytes" are one page of machine language routines snatched from the Autostart ROM. One man's program is another man's scrambled bytes.) Before you execute it, study the assembler listing. It uses a couple of tricks. Lines 1070 and 1080 are **equates.** ".EQ" is a "pseudo op" , an assembler directive that makes it internally associate the name "COUNTR" with the number $FA. In writing the source program, you use the name, not the number. Notice that the assembler didn't generate any bytes in response to .EQ statements.

```
                    1000 * BUBBLE SORT
                    1010 *
                    1020 *
                    1030 * SORTS DATA FROM $A00 - $AFF
                    1040 * SMALLEST VALUES TO TOP
                    1050 *
                    1060 *
                    1070 COUNTR .EQ $FA
                    1080 TABLE  .EQ $A00
                    1090 *
                    1100 *
0800- A9 FF         1110 START  LDA #$FF
0802- 85 FA         1120        STA COUNTR     COUNT = 255
0804- A0 00         1130 LOOP   LDY #$00       Y IS PONTER
0806- B9 00 0A      1140 LOOP2  LDA TABLE,Y
0809- C8            1150        INY
080A- D9 00 0A      1160        CMP TABLE,Y
080D- B0 0D         1170        BCS NOSWAP
                    1180 *
                    1190 * SWAP TABLE,Y AND TABLE,Y-1
                    1200 *
080F- AA            1210        TAX            SAVE IT
0810- B9 00 0A      1220        LDA TABLE,Y
0813- 88            1230        DEY
0814- 99 00 0A      1240        STA TABLE,Y
0817- C8            1250        INY
0818- 8A            1260        TXA            RESTORE IT
0819- 99 00 0A      1270        STA TABLE,Y
081C- C4 FA         1280 NOSWAP CPY COUNTR
081E- D0 E6         1290        BNE LOOP2
0820- C6 FA         1300        DEC COUNTR
0822- D0 E0         1310        BNE LOOP
0824- 60            1320        RTS
```

Line by line:

1110 - 1120 — Initialize counter to $FF. Counts how many passes we must make through the list. We're done when this is reduced to zero.

1130 — Starting point of outer loop. Puts us at the bottom of the list for the start of each pass. The Y register is the pointer.

1140 — Starting point of the inner loop, where we work our way up, pair by pair, until we reach the top.

1140 - 1270	The actual comparing and swapping. Test each consecutive pair. If the byte with the lower address is smaller than the addressed byte, swap them.
1280	Have we gone all the way through the list yet? Remember, we don't need to go any higher than we've already done passes.
1300	An entire pass has been completed. If we've done 256 passes, we're done. Otherwise, make another pass.

Now execute it. Unless you've really got a handle on every step, use the simulator for a couple of comparisons. Understanding the comparison step requires understanding the borrow flag. Take a whole day if you must to get it down pat, but do it, once and for all. This program (all $25 bytes of it) takes less than a second to sort the list in 6502 mode. Not bad for 32,000 comparisons and 16,000 swaps.

THE GREAT DRAW LINES AND BEEP PROGRAM

This last program, at a whopping $130 bytes, is by far the longest and most complex in this manual. **Beep-a-Sketch** (don't like the name? Send us a better one) draws a free running line on the screen. You control where the line goes with the I-J-K-M diamond. Accompanying the line is a tone that goes up when the line goes up and down when the line goes down. The arrow keys control how fast the line moves (left arrow slows it down, right arrow speeds it up). Pressing "C" (Clear) erases the screen. Escape ends the program. For a change, we're going to run this program before we try to understand it. Bload it and GO (again, use a JSR at $300 to fire it up).

Now that you've had your fun, let's figure out how these $130 bytes made it happen. Consult the assembler listing of the program at the end of this chapter as we work through it.

APPLE HIRES GRAPHICS FUNDAMENTALS

In high resolution mode (also known as **Root Beer**, or **Hires**, graphics), the Apple programmer works with a grid of 280 horizontal points by 192 vertical points. The upper left corner is point 0,0, the lower right corner 279, 191. Each dot on the screen corresponds to a bit in memory. Bits that are set display as white. Bits that are reset display as black. Hires page one resides in memory from $2000 - $3FFF. This "bit mapping" technique is discussed in detail on pages 19 - 22 of the **ARM**.

Applesoft hires graphics programs don't have to worry about the nuts and bolts of addresses and bits. Executing the Basic command HPLOT 123,16 causes a hard working machine language routine in ROM to locate the correct address in memory (somewhere between $2000 and $3FFF) and set whatever bit needs setting to light up grid position 123,16.

The problem faced by the machine language programmer is how to take an X,Y coordinate (the natural way to express an object's position on a grid), and locate the exact address and bit position required to plot the point. It just so happens that the whereabouts of the Applesoft subroutine that performs the nitty gritty plotting is known, and may be called by a machine language program. Beep-a-Sketch makes extensive use of Applesoft graphics subroutines, and was therefore much easier to write (and shorter) than it would have been if we had had to write our own routines to do these things.

APPLESOFT HIRES GRAPHICS ROUTINES

HLIN ($F53A) Draws a line from the last plotted point or line destination to the horizontal and vertical coordinates passed in the registers.

```
Horizontal MSB = X
Horizontal LSB = A
Vertical       = Y
```

HPLOT ($F457) Plots a dot at the horizontal and vertical coordinates passed in the registers. Note that the registers are used differently from HLIN.

```
Horizontal MSB = Y
Horizontal LSB = X
Vertical       = A
```

HCLR ($F3F2) Clears the screen to black. No parameters.

SETCOL ($F6EC) Sets the hires color. In the X register, pass it the same 0 - 7 values of the Applesoft HCOLOR command.

HGR ($F3E2) Clear hires page 1, mixed text and graphics. Same as Applesoft HGR command.

Lines 1130-1180 equate appropriate words with the addresses of the Applesoft graphics subroutine. Lines 1220-1250 do the same for some

I/O addresses. (NOMIX is an address dependent switch that makes hires graphics display all the way to the bottom, instead of sharing space with four lines of text.)

START is Beep-a-Sketch's entry point. It is convention to have the first address in a program the starting point (DOS's BRUN command assumes this). This is the first line of the source program that causes the assembler to generate any code.

The first instruction jumps around several bytes in memory that are used as data and variables. The **.DA** (data) psuedo op tells the assembler to set aside a byte, assign it a label, and initialize it to a given value. Once we've allocated space for a variable and assigned it a name, we can use it almost the same way we use variables in Basic.

.HS (hex string) is a psuedo op that causes the assembler to produce several consecutive bytes initialized to the values in the operand. The label DELTA is the address of the first of the five bytes in the table.

BEEP A SKETCH VARIABLES

VP – Vertical Position. Current Y coordinate.

HP – Horizontal Position. Current X coordinate.

HINDX and **VINDX** (Horizontal and vertical indexes). These values control how new VP and HP's will be calculated with each pass through the loop.

SPEED – Used as the duration parameter in the BEEP subroutine. Controls how fast the program runs by determining how long BEEP lasts.

MAIN LOOP

Like all good machine language programs, the main loop is a simple series of subroutines. We can understand the overall operation of the program without knowing exactly how each subroutine works.

The INIT subroutine does a couple of things that only need doing once, such as clearing the hires display and setting the plotting color to white.

DRWLIN is the working heart of Beep-a-Sketch. First we use the Applesoft HPLOT routine to draw a white dot at the current **HP** and **VP** values (initialized to 125,80).

Next, calculate a new value for **HP** and **VP** with subroutines NEWH and NEWV. We'll discuss these routines in a minute; for now, all we care is that **HP** and **VP** return from this routine with (slightly) different values. Now, use the Applesoft HLIN routine to draw a line connecting the dot we drew a second ago to the new **HP,VP** value.

When the line is drawn, we return to the main loop, and do a beep. This version of BEEP takes its duration value from the variable **SPEED**, and its pitch from **VP**. The less **VP** is (closer to the top of the screen) the higher the pitch.

Next, the main loop checks the keyboard to see if there's a request to deal with. Note that this program doesn't check the keyboard repeatedly like ASCII organ did. ASCII organ would wait forever for a keypress. Beep-a-Sketch checks it, and if there wasn't a key pressed, jumps to the top of the loop.

If there is a key pressed, we call subroutine KEYPRS that checks to see what key it is, and if one of the keys that control the program, performs the requested function. Eventually we get back to the main loop and the process repeats.

THE HARD STUFF—NEWH/NEWV

The NEWH/NEWV routines calculate new values for HP and VP based on three facts:

1) The values of HP and VP coming in

2) The maximum and minimum vertical and horizontal positions allowed. (2 < HP < 250, 2 < VP < 180)

3) The current horizontal and vertical indexes.

First, HP is checked for range. We don't let it get larger than 250 or less than 2. Beep-a-Sketch simplifies things by keeping X to a value that can be represented by one byte, at the cost of not using the full screen width. If the range tests pass (let's say they did), we calculate a new HP according to HINDX. This calculation is Beep-a-Sketch's stickiest wicket. HINDX controls how much larger or smaller HP will be when we leave this subroutine. HINDX can range from 0 - 4, and is used to index one of the five bytes in the data table DELTA

(i.e., DELTA is a label equivalent to the address of the first byte in the table).

DELTA contains the two's complement form of the numbers -2, -1, 0, 1, and 2, in that order. An HINDX of 0 causes the ADC DELTA,X statement in line 2000 to add the two's complement form of -2, $FE, to HP. Because of the magic of two's complement, this has the effect of **reducing HP by two.** A different HINDX would index a different byte of the table, and a different change for HP. The possible values of HINDX are 0-4, corresponding to changes of -2, -1, 0, 1, and 2. Once we've range tested and calculated a new HP, NEWH is done.

NEWV is identical to NEWH, only with different range testing values.

KEYPRS tests for the eight keys that are defined to mean something. If one of the control diamond keys was pressed, we must increase the tendancy of the line to move in the selected direction, **by altering the appropriate index either up or down.** If J is pressed, for example, we want to increase the tendency of the line to go left; going left means smaller X values. We **decrease** HINDX, so that in the next DRWLIN, we'll calculate an HP that is **smaller** than it was previously. A small HINDX indexes a negative value to add to HP. The table DELTA is only five elements long, though, so we can't let HINDX get below zero. Pressing K increases HINDX (but no larger than 4); I decreases VINDX; M increases VINDX.

If one of the arrow keys was pressed, we change the variable SPEED that controls how long BEEP lasts. There is no range testing on SPEED—we let it roll over.

If "C", wipe the screen clear with the Applesoft call HCLR.

If escape, we abort by returning all the way out of the program. Two pulls of the accumulator produce a stack with the return address of the next most recent subroutine on top; RTS now puts us back where we started, either in the monitor, or back in TVC, depending how we got here. This is similar to Basic's POP instruction.

Beep-a-Sketch is a whole fistful of programming tricks, no doubt about it. Take it a subroutine at a time, top down, in trying to understand it. Run it with the simulator at first; Use GO to run time consuming subroutines like beep, or to skip over the mysteries of Applesoft's hires graphics routines. These routines are one place where we are willing to accept a gift without worrying about the particulars. They work—just learn how to use them. Use the simulator for the tricky steps of KEYPRS and NEWH, NEWV.

BEEP-A-SKETCH ASSEMBLER LISTING

```
                    1000 * BEEP A SKETCH
                    1001 *
                    1002 * BY GEORGE CONSAR
                    1004 *
                    1010 *
                    1020 * I-J-K-M DIAMOND CONTROLS
                    1030 * MOVEMENT OF LINE
                    1040 *
                    1050 * "C" CLEARS SCREEN
                    1060 * <ESC> ABORTS PROGRAM
                    1070 *
                    1080 * ARROW KEYS CONTROL SPEED
                    1090 *
                    1100 *
                    1110 * APPLESOFT SUBROUTINES
                    1120 *
                    1130 HLIN   .EQ $F53A
                    1140 HPLOT  .EQ $F457
                    1150 HPOSN  .EQ $F411
                    1160 HCLR   .EQ $F3F2
                    1170 SETCOL .EQ $F6EC
                    1180 HGR    .EQ $F3E2
                    1190 *
                    1200 * I/O LOCATIONS
                    1210 *
                    1220 KEYBD  .EQ $C000
                    1230 KEYSTB .EQ $C010
                    1240 SPKR   .EQ $C030
                    1250 MONIX  .EQ $C052
                    1260 *
                    1270 *
0800- 4C 0D 08      1280 START  JMP SKETCH
                    1290 *
                    1300 *
                    1310 * VARIABLES
                    1315 *
                    1320 *
0803- 03            1330 HINDX  .BA #3        CHANGE OF X
0804- 03            1340 VINDX  .BA #3        CHANGE OF Y
                    1345 * PLOT A POINT AT HP,VP
0805- 5A            1350 VP     .BA #90       VERTICAL POSITION
0806- 7D            1360 HP     .BA #125      HORIZONTAL POSITION
0807- 81            1370 SPEED  .BA #$81      LENGTH, DELAY LOOP
0808- FE FF 00
080B- 01 02         1380 DELTA  .HS FEFF000102 -2,-1,0,1,2
                    1390 *
                    1400 **********************
                    1410 *                      *
                    1420 * MAIN LOOP           *
                    1430 *                      *
                    1440 *                      *
080B- 20 24 08      1450 SKETCH JSR INIT        *
0810- 20 30 08      1460 LOOP   JSR DRWLIN      *
0813- 20 16 09      1470        JSR BEEP        *
0816- AD 00 C0      1480        LDA KEYBD       *
0819- 10 F5         1490        BPL LOOP        *
081B- 2C 10 C0      1500        BIT KEYSTB      *
081E- 20 AB 08      1510        JSR KEYPRS      *
0821- 4C 10 08      1520        JMP LOOP        *
                    1530 *                      *
                    1540 **********************
                    1550 *
```

```
                1560 *
0824- 20 E2 F3 1570 INIT   JSR HGR       INITIALIZE HIRES GRAPHICS
0827- 2C 52 C0 1580        BIT HOMIX     FULL SCREEN GRAPHICS
082A- A2 03    1590        LDX #3
082C- 20 EC F6 1600        JSR SETCOL    COLOR = WHITE
082F- 60       1610        RTS
                1620 *
                1630 * DRAW LINE SUBROUTINE
                1640 *
                1650 * CALCULATE NEW HP AND VP
                1660 * DRAW LINE TO NEW HP,VP
                1670 *
0830- AE 06 08 1680 DRWLIN LDX HP
0833- A0 00    1690        LDY #00
0835- AD 05 08 1700        LDA VP
0838- 20 57 F4 1710        JSR HPLOT     LOT A POINT AT OLD HP,VP
083B- 20 4B 08 1720        JSR NEWH      CALCULATE NEW HORIZONTAL POS
083E- 20 7C 08 1730        JSR NEWV      CALC NEW VERTICAL POS
0841- AD 06 08 1740        LDA HP        A = LSB OF HORIZONTAL POS
0844- A2 00    1750        LDX #0        X = MSB OF HORIZONTAL POS
0846- AC 05 08 1760        LDY VP        Y = VERTICAL POS
0849- 20 3A F5 1770        JSR HLIN      DRAW LINE
084C- 60       1780        RTS
                1790 *
                1800 * CALCULATE NEW HP
                1810 *
084D- A9 FA    1820 NEWH   LDA #250       MAXIMUM H. POSITION
084F- CD 06 08 1830        CMP HP
0852- B0 0B    1840        BCS NEWH1     NOT TOO BIG YET
0854- 8D 06 08 1850        STA HP        ELSE, MAKE IT 250
0857- A9 01    1860        LDA #1        AND CHANGE MOTION INDEX
0859- 8D 03 08 1870        STA HINDX
085C- 4C 6E 08 1880        JMP NEWH2
                1890 *
085F- A9 02    1900 NEWH1  LDA #2        MIN H POS = 2
0861- CD 06 08 1910        CMP HP        HP TOO SMALL?
0864- 90 08    1920        BCC NEWH2     NAH...
0866- 8D 06 08 1930        STA HP        YES, MAKE IT 2
0869- A9 03    1940        LDA #3        AND SET NEW H INDEX
086B- 8D 03 08 1950        STA HINDX
                1960 *
086E- AD 06 08 1970 NEWH2  LDA HP        RANGE TESTING OVER
0871- AE 03 08 1980        LDX HINDX     READ XTH ENTRY OF DELTA TABLE
0874- 18       1990        CLC           AND ADD IT TO HP (MAY BE NEG OR POS)
0875- 7D 08 08 2000        ADC DELTA,X
0878- 8D 06 08 2010        STA HP
087B- 60       2020        RTS
                2030 *
                2040 * CALCULATE NEW VP
                2050 *
087C- A9 B4    2060 NEWV   LDA #180       180=MAX V
087E- CD 05 08 2070        CMP VP        TOO BIG?
0881- B0 0B    2080        BCS NEWV1     NOPE
                2090 *
0883- 8D 05 08 2100        STA VP        YES, IT WAS
0886- A9 01    2110        LDA #1        SET NEW VINDX
0888- 8D 04 08 2120        STA VINDX
088B- 4C 9D 08 2130        JMP NEWV2
                2140 *
088E- A9 02    2150 NEWV1  LDA #2        2=MIN V
0890- CD 05 08 2160        CMP VP        TOO SMALL?
0893- 90 08    2170        BCC NEWV2     NO...
                2180 *
0895- 8D 05 08 2190        STA VP
0898- A9 03    2200        LDA #3        SET NEW VINDX
089A- 8D 04 08 2210        STA VINDX
```

```
                2220 *
089B- AD 05 08   2230 MENU2   LDA VP
08A0- AE 04 08   2240         LDX VINDX
08A3- 18         2250         CLC
08A4- 7B 08 08   2260         ADC DELTA,X   SAME TRICK AS IN MENU
08A7- 8D 05 08   2270         STA VP
08AA- 60         2280         RTS
                2290 *
                2300 * KEYPRS -- HANDLE HUMAN'S INPUT
                2310 *
08AB- C9 C9      2320 KEYPRS  CMP #$C9      1 KEY?
08AD- D0 09      2330         BNE KEY2      NO
                2340 *
08AF- AD 04 08   2350         LDA VINDX
08B2- F0 03      2360         BEQ KEY1      ALREADY ZERO?
08B4- CE 04 08   2370         DEC VINDX     NO,DEC IT
08B7- 60         2380 KEY1    RTS
                2390 *
08B8- C9 CD      2400 KEY2    CMP #$CD      M KEY?
08BA- D0 0B      2410         BNE KEY4      NO
                2420 *
08BC- AD 04 08   2430         LDA VINDX
08BF- C9 04      2440         CMP #04       ALREADY 4?
08C1- F0 03      2450         BEQ KEY3      YES,RTS
08C3- EE 04 08   2460         INC VINDX     NO,INC IT
08C6- 60         2470 KEY3    RTS
                2480 *
08C7- C9 CA      2490 KEY4    CMP #$CA      J KEY?
08C9- D0 09      2500         BNE KEY6      KEEP LOOKING
                2510 *
08CB- AE 03 08   2520         LDX HINDX
08CE- F0 03      2530         BEQ KEY5      ALREADY 0 ?
08D0- CE 03 08   2540         DEC HINDX     NO, DEC IT
08D3- 60         2550 KEY5    RTS
                2560 *
08D4- C9 CB      2570 KEY6    CMP #$CB      K KEY?
08D6- D0 0B      2580         BNE KEY8      HARDLY
                2590 *
08D8- AD 03 08   2600         LDA HINDX
08DB- C9 04      2610         CMP #04       HINDX=4?
08DD- F0 03      2620         BEQ KEY7      YES,RTS
08DF- EE 03 08   2630         INC HINDX     NO, INC IT.
08E2- 60         2640 KEY7    RTS
                2650 *
                2660 * NOW CHECK FOR ARROW KEYS--
                2670 * (AREN'T HUMANS A LOT OF TROUBLE?)
                2680 *
08E3- C9 88      2690 KEY8    CMP #$88      BACK ARROW?
08E5- D0 0E      2700         BNE KEY9
                2710 *
08E7- AD 07 08   2720         LDA SPEED
08EA- C9 F0      2730         CMP #$F0
08EC- F0 06      2740         BEQ KEY10
08EE- 18         2750         CLC
08EF- 69 10      2760         ADC #$10
08F1- 8D 07 08   2770         STA SPEED
08F4- 60         2780 KEY10   RTS
                2790 *
08F5- C9 95      2800 KEY9    CMP #$95      RIGHT ARROW?
08F7- D0 0E      2810         BNE KEY12
08F9- AD 07 08   2820         LDA SPEED
08FC- C9 10      2830         CMP #$10
08FE- F0 06      2840         BEQ KEY11
0900- 38         2850         SEC           CLEAR BORROW!!!!
0901- E9 10      2860         SBC #$10
0903- 8D 07 08   2870         STA SPEED
0906- 60         2880 KEY11   RTS
```

```
                2890 *
                2900 * TWO POSSIBILITES LEFT, C AND <ESC>
                2910 *
0907- C9 C3     2920 KEY12  CMP #$C3      C?
0909- D0 04     2930        BNE KEY13
090B- 20 F2 F3  2940        JSR HCLR      ERASE SCREEN
090E- 60        2950        RTS
                2960 *
090F- C9 9B     2970 KEY13  CMP #$9B      <ESC>?
0911- D0 02     2980        BNE KEY14
0913- 68        2990        PLA           THESE PULLS
0914- 68        3000        PLA           MAKE RTS GO ALL THE WAY BACK
0915- 60        3010 KEY14  RTS
                3020 *
                3030 *
                3040 * SAME OLD BEEP ROUTINE
                3050 * BUT WITH VARIABLE DURATION VALUE
                3060 *
                3070 *
0916- AC 07 08  3080 BEEP   LDY SPEED
0919- AE 05 08  3090 BEEP1  LDX UP
091C- CA        3100 BEEP2  DEX
091D- D0 FD     3110        BNE BEEP2
091F- 2C 30 C0  3120        BIT SPKR
0922- 88        3130        DEY
0923- D0 F4     3140        BNE BEEP1
0925- 60        3150        RTS
```

SYMBOL TABLE

HLIN	F53A	HPLOT	F457	HPOSN	F411
HCLR	F3F2	SETCOL	F6EC	HGR	F3E2
KEYBD	C000	KEYSTB	C010	SPKR	C030
HOMIX	C052	START	0800	HINDX	0803
VINDX	0804	UP	0805	HP	0806
SPEED	0807	DELTA	0808	SKETCH	080D
LOOP	0810	INIT	0824	DRHLIN	0830
MENH	084D	MENH1	085F	MENH2	086E
MENV	087C	MENV1	088E	MENV2	089D
KEYPRS	08AB	KEY1	08B7	KEY2	08B8
KEY3	08C6	KEY4	08C7	KEY5	08D3
KEY6	08D4	KEY7	08E2	KEY8	08E3
KEY10	08F4	KEY9	08F5	KEY11	0906
KEY12	0907	KEY13	090F	KEY14	0915
BEEP	0916	BEEP1	0919	BEEP2	091C

16.
Where Do I Go From Here?

Buy an assembler. Quick. Now that you know what an assembler can do, never again waste time looking up opcodes or calculating relative branches.

A good value is the Editor-Assembler of Apple's **DOS Tool Kit.** Not only do you get a serviceable text editor and assembler, they throw in two products that no Apple programmer should be without: The High Resolution Character Generator, and the Apple Programmer's Assistant. If you don't have the Tool Kit already, run, don't walk, to your nearest Apple dealer and get one. (No charge for the promo, Steves.)

The non-trivial programs of the PROG series (i.e., longer than six bytes) were done with the **SC Assembler II,** one of the earliest Apple assemblers. Although not as powerful as the Tool Kit Assembler, it's a good choice for beginners, because in many ways (like editing and use of DOS) it acts like Basic. It's available from your local software dealer, or from the publisher:

S-C Software Corporation
2331 Gus Thomasson, Suite 125
Dallas, Texas 75228

Other assemblers with good reputations are **LISA** from On-Line Systems and Southwestern Data Systems' **Merlin.**

One of the things you get when you buy a 16K RAM card is access to Apple's **mini-assembler.** It gets loaded along with integer Basic when you boot the system master. A mini-assembler offers convenience between raw hex and a full assembler. For programs 20-30 bytes long nothing can beat it. Programs longer than about 50 bytes become a real hassle to keep straight. If you couldn't justify the cost of a RAM card on the basis of integer Basic alone, think seriously about getting one to acquire the mini-assembler.

Whichever assembler you decide on, don't expect to be doing great things with your new assembler the first day you peel off the shrink wrap. With all this power come a host of new things to learn. The editing commands. What you do to delete, add, and change lines in your source program. You may have to unlearn some editing commands you learned to run a word processing program.

You'll have to learn your assembler's "pseudo ops", special mnemonics that are not 6502 instructions, but commands for the assembly process to follow. Such as turning the printer on for page three of a listing and off on page five. Or determining where in memory a source program will be assembled to run. Expect to work as hard learning to effectively use an assembler as you did getting this far in machine language.

I used my assembler to write a program but it doesn't work and I don't know why. Programs **never** work the first time, especially machine language programs. A machine language program that takes a wrong turn can go a lot of places and do a lot of bad things in a hurry. To debug it you can either use the time-honored Apple monitor techniques, or The Visible Computer, or both.

THE APPLE MONITOR

While not as flashy or friendly as TVC, the Apple monitor has everything you need to debug and perform simple patches to machine language programs. It is well documented in Chapter 3 of the **ARM.**

Let's walk through a session of using the Apple monitor (hereinafter referred to as the monitor). Boot your computer with an ordinary 3.3 disk. When Basic signs on tell it to get lost and put you in the monitor with: **CALL -151.** The monitor prompt is the asterisk.

DOS is still active; although you can't use Basic commands (RUN, LIST) because they don't make sense to the monitor, you may still use DOS commands (CATALOG, BLOAD). The same cursor movement-editing techniques may be used, also.

To execute a program you'll need to either write it on the spot, or load it from disk at the address you specify with DOS's bload command. (Consult the DOS manual for particulars.)

Once loaded, you can list it with the monitor's disassembly command, "L" (how original). The Apple **L** command does not need or want a space between the address and the "L". While this difference from TVC syntax may take a little getting used to, be consoled by the fact that not only do you get 20 instructions at a time instead of TVC's 5, you

get them lightning fast. Vivid proof of the speed difference between Basic and machine language.

You can execute your program by typing the address of the program's entry point and a "G". Not GO, but G. This is not a buffered, protected execution, but the real McCoy. No polite stopping when you use invalid opcodes, or access I/O addresses that make things go crazy. You will return to the monitor when your program executes a final RTS.

How can you run your program in pieces so you can see where it is going wrong? At one time all Apples had the monitor functions **Step** and **Trace.** I almost hate to mention their existence, because your Apple probably doesn't have them. They were purged in the transition of 1979 from the "Old Monitor ROM" to the Auto-start ROM.

Don't get me wrong–the new ROM has some good stuff in it, like automatic boot on powerup; staying in Basic when you press reset; the ability to stop scrolling with Ctrl-S (like a cursor, another thing you don't appreciate until you don't have it); enhanced editing capability. But machine language programmers must make do without the debugging commands **Step** (execute one instruction, display the registers and return to the monitor) and **Trace** (execute one step after another).

Fortunately, BRK can be used to do the same thing. As we learned in Chapter 15, BRK causes a jump to $FA62, a ROM routine that displays the registers and exits to the monitor. By judiciously placing $00's in your program you can find out if it's getting to certain points of your program, and if so, what values the registers have achieved. You'll end up using BRK in a monitor debugging session the same way that the TVC simulator does in non-master mode, as a signal to stop execution.

The monitor also has provisions for simple hex math, moving memory, and numerous other tricks, all documented in the **ARM.**

BASIC AND MACHINE LANGUAGE: SHARING THE WORK

We said way back in Chapter 1 that the smart Apple programmer doesn't use machine language unless he has to. Even then, he first tries to get the job done with a hybrid program: Basic providing the main framework and machine language for the part that isn't fast enough in Basic. To accomplish sharing requires an understanding of what Basic is and isn't.

WHAT IS BASIC?

Applesoft is a long (**real** long) 6502 machine language program that resides in ROM from $D000 - $F800. Physically, these 10K bytes are stored in five fat 24 pin IC's just the keyboard side of the 6502. Applesoft hogs, and rightfully so, about half of the addresses in page zero for its variables. If it didn't, Basic programs would run even slower. Pages 140 and 141 of the **Applesoft II Basic Programming Reference Manual** (the green one), give a detailed breakdown on what locations are used for what.

Applesoft creates an environment in which the programmer works with concepts instead of registers and addresses. Where you can copy a formula like **X = SIN (2*Y)** almost straight out of a math book, without worrying how sines are calculated. Where you can say **X = Y + Z** without considering where in memory to store variable X. Applesoft helps to bridge the enormous gap between the English language and standard mathematical representation, and the 8 bit, 56 instruction world of the 6502.

If you learned one thing in this book, it's that a 6502 can no more understand:

1000 INPUT "ENTER YOUR NAME ";A$

than it can play chess or dance the two-step. 6502 **programs** can be written to do these things, but the 6502 just rolls, shifts, jumps, adds, subtracts, etc...

A Basic program is an elaborate data table constructed and maintained by the machine language program Applesoft. The table begins at $801 and works up. Actually, it works up from $801 and down from high memory; you run out of space when the two parts meet. When you run a Basic program, at no time does the 6502 JMP or JSR to any of the data in this table. Instead Applesoft executes a series of subroutines in ROM that first decode, ("parse") and then execute, this data.

The appendices of the Applesoft Manual contain information about how Applesoft organizes its data; particularly how variables are stored. In general, however, Applesoft is not as well documented as we'd like, at least not by Apple. Trade secrets, that kind of thing.

If you read enough computer magazines, you will find articles on how Applesoft does things, painstakingly mined by Applesoft cultists through hours of careful disassembly. Of particular interest are the locations of subroutines that perform functions your machine language programs can tap. Who needs to write a floating point square root routine or a hi-resolution line drawing routine when there's already one there?

HOW TO ORGANIZE BASIC AND MACHINE LANGUAGE

To run a machine language subroutine from Basic, use Applesoft's CALL instruction. Give it the decimal version of your routine's starting address. To call a subroutine at $280, use **CALL 640.** When it executes a final RTS, Basic will pick up execution with the statement immediately after the call.

Where do I put them? If your machine language routines are fairly small, you can hide them in the first $D0 bytes of page 3, left vacant for just this purpose. CALL 768 appears in so many programs it's almost a bonafide Applesoft command. Page 2 is used by the monitor's GETLN subroutine (used by Applesoft to get lines from the keyboard during editing and INPUT statements–see page 33 of the **ARM**) as a keyboard buffer; if you're not planning on using INPUT statements to collect answers that long, you can use the upper regions of this page safely.

There's no room for **programs** in page zero, but a few spaces are left unused by DOS, Applesoft, and the monitor, and you are welcome to them for your important variables and pointers.

If you need more room than what's available in pages two and three, you'll need to relocate Basic. Applesoft defaults to $801 for program storage, but this can be changed. Locations $67 and $68 point to the beginning of space used for Applesoft. If you change the numbers stored there, the **next** program loaded will run from the new address.

This short program sets Basic's start-of-memory pointer to $4001. This is a good spot, because not only does it free up about 6K of space for programs, it leaves hires graphics page 1 safely underneath the doings of Applesoft. You still have 22K of Basic program space to work with, and that's plenty, especially considering that you may chain back and forth between different programs.

```
100 POKE 104,64
110 POKE 16384,0
110 PRINT CHR$(4)"RUN PROGRAM"
```

For some reason (probably a good one), Applesoft goes crazy if it doesn't find a zero loaded just below the start of programs.

Parameter Passing. If the machine language routine needs little or no additional information when it is called, you can get by with POKEing a few values into memory (naturally, at the place where the machine language subroutine knows to look for its data.) If the application requires a lot of data transfer between Basic and machine language, you can have the subroutine act on the desired Applesoft variable(s) directly. This is more difficult, as it requires a thorough understanding of Applesoft's variable handling.

THE END OF THE ROAD

That's it for the tutorial part of the Visible Computer. Obviously, you haven't learned everything there is to know about machine language. Like any discipline, learning machine language involves more than reading one book. Here are three sure-fire ways to improve your programming skills:

1. Read good books. Three good ones are listed below. There are good ones I've left out, but beware of the Judging-by-the-Cover syndrome in programming books. There are some **bad** ones out there.

2. Study other people's assembly language programs. The monitor listing in the **ARM** is a rich (too rich, sometimes) source of ways to get things done in machine language.

3. Give yourself projects. Pick a task that seems suited to your capabilities (although sometimes the most innocent projects prove to be bottomless pools of complications). Maybe you could alter Beep-a-Sketch to have a no-beep mode. Or to use the whole width of the screen. When you lick one project, move to a more difficult one.

SUGGESTED READING

Assembly Lines: The Book – A Beginner's Guide to 6502 Programming on the Apple II

By Roger Wagner. Softalk Publishing, 11021 Magnolia Boulevard, North Hollywood, CA 91601

A clear, friendly presentation full of small programs that do a lot. Extensive use of monitor subroutines.

Programming a Microcomputer: 6502

By Claxton Foster. Addison-Wesley Publishing Company, Inc.

A funny little book, with some of the worst diagrams, but best descriptions anywhere. You'll need to read between the lines of this book somewhat, as its target vehicle is not the Apple, but the KIM single board computer. It did a lot for me, though.

6502 Programming

by Rodnay Zaks. SYBEX, Inc.

A detailed reference guide, with extensive discussions of signed numbers, and demonstration programs implementing various arithmetic and sorting problems.

Appendix A
Behind the Scenes of TVC

TVC Memory Map

```
-------------------------------------------------
$BFFF:
     : DOS
$9AA6:
     :  TVC (Applesoft Basic program)
$4000:
     : Hires Page 1--Main Display
$2000:
     : TVC Decode Tables
     :
$1A00: TVC Machine Language Routines
     :
     :  DOS Tool Kit Hi-res Character Generator
$0EFF:
     : JSR Handler
$0E00:
     : TVC Stack
$0D00:
     : TVC Zero Page
$0C00:
     : 1K User Memory
$0800:
     :  Display Buffer
$0400:
     : Page 3; $3D0-$3FF = DOS and monitor vectors
$0300:
     : GETLN buffer. Not used by TVC.
$0200:
     : 6502 Stack
$0100:
     :  6502 Zero Page--TVC uses locations $06-$09, $FE-$FF
$0000:
     -------------------------------------------------
```

To protect itself from user programs in non-master mode, The Visible Computer maintains separate zero and stack pages. You can verify this by comparing reads of $0000-$01FF and $0C00-$0DFF.

The Visible Computer: 6502 is a machine language and Applesoft Basic hybrid. The Basic part does 95% of the work (it would have been almost as easy to write The Visible Computer: 6502 on the TRS-80, a Z-80 machine). Machine language routines are used primarily to calculate the result of arithmetic, logical, and shift instructions. Basic is singularly unsuited for bitwise manipulations.

THE GO COMMAND

The sequence of events in a GO command: The address of the subroutine is poked into locations $EO7 (LSB) and $E08 (MSB). The zero page share flag goes to $E86. This flag controls whether or not a swap is made of the real zero page and the TVC zero page before giving your program control. Basic then calls $E00, the Go handler:

```
0E00-   20 10 0E    JSR   $0E10
0E03-   20 26 0E    JSR   $0E26
0E06-   20 FF FF    JSR   $FFFF
0E09-   20 35 0E    JSR   $0E35
0E0C-   20 10 0E    JSR   $0E10
0E0F-   60          RTS
```

$E10 swaps the TVC and real zero pages (if the swap flag is set). $E26 loads the registers with values that were in the TVC registers. We then call the user's program, and when it returns, save the registers, and swap the zero pages back.

If you swap zero pages, (i.e., execute a GO with TVC in zero page noshare mode), be aware that many monitor routines are going to go nuts now that they have a zero page full of zeros or random data where their variables should be. You may want to prepare a dummy zero page for just such an occasion. On the other hand, if you share the real zero page, and don't make the swap, you run the risk of stomping on memory locations that Basic, DOS, and the monitor need to be healthy. Such is the life of a machine language programmer.

There is no way to share the stack. The simulator always uses the bogus stack page at $D00. A subroutine passed to the 6502 via the GO command will use the real 6502 stack for all pushes, pulls, and subroutine addresses. Any data you write to the stack page during a GO will not appear to be there when you get back to TVC.

DISCLAIMER

The Visible Computer: 6502 is a tool for teaching machine language programming; a secondary function is the debugging of 6502 programs. **It is not intended to be a rigorous copy of a 6502's internal workings.** It **does** execute all 151 defined opcodes correctly, down to the JMP (IND) bug. It may, however, arrive at identical results through different mechanisms. The term "microstep" has a conceptual kinship to microcode, but any similarity between the real working microcode of a 6502 and the eight TVC microsteps is coincidental.

Appendix B
ASCII Character Set

		Most Significant Bits							
HEX		0	1	2	3	4	5	6	7
	BINARY	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	SPACE	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	¦
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

Codes $00- $1F are **control characters.** They have no printed equivalent but rather serve to "control" a device. Sending a printer ASCII code $46 makes it print an "F". Sending it a form feed character ($0C) makes it scroll to top of form, without printing anything.

Control codes $01 through $1A can be generated by the Apple keyboard by depressing the Control key in conjunction with one of the 26 alphabetic keys. Some of the important control codes have specific keys dedicated to them; the return key generates the same code as Ctrl-M, $0D. The following control characters are the ones most likely to be encoutered using Apple computers:

CODE	MNEMONIC	OPERATION
$07	BEL	Sound Bell (or beep)
$08	BS	Backspace Cursor
$0A	LF	Line feed
$0C	FF	Form feed
$0D	CR	Carriage return
$1B	ESC	Escape

The Apple left arrow key generates a backspace, code $08. The right arrow key generates a NAK, ($15).

The Apple keyboard cannot generate the codes for lower case letters.

Normal Apple protocol is to store characters with bit 7 set.

The codes for lower case letters are the same used for upper case with bit 6 set. The lower four bits of the ASCII code for the numbers is equivalent to their value. If you mask off the high order four bits with AND #$0F, you turn an ASCII value into a binary number.

This page intentionally left blank

Appendix C
Monitor Commands Reference

Monitor mode is indicated by the "#" (pound sign) prompt on the last line of the display. This indicates TVC's readiness to accept one of the 21 commands that control it.

Monitor commands have the general form:

<command> [argument1] [argument2]

You **must** separate a command and its arguments by one or more spaces. You **must not** use spaces within a command or argument.

This list of TVC monitor commands uses the following conventions:

<address> A number **valid in the current monitor base** that is greater than or equal to zero, and less than 65536.

<value> A number **valid in the current base,** where n = 0 256.

<register> An on-screen register. They are: DL, DB, IR, A, S, P, X, Y, PC, AD, RAMA, RAMD

<filename> A valid DOS file name, without embedded spaces. "TESTFILE"; "PROGRAM1"

Slashes "/" are used to indicate equivalent command parameters.

Square brackets "[]" indicate optional parameters.

BASE

Change display or monitor base.

Syntax: **BASE <register> HEX/BIN/DEC**

This command controls how numbers will be both displayed on the screen and or interpreted when entered in the monitor. In place of **<register>** one may use:

ALL change base of **all** registers.

MEM change base of mem display.

MON change monitor base.

Example: **BASE PC BIN**

STEP

Set simulator step mode.

Syntax: **STEP 0/1/2/3**

This function sets the stepping rate of the 6502 simulator. The effect of each step value is summarized below.

Value	Function
3	Pause at various key points in each instruction. Return to monitor when instruction complete.
2	No pause during instruction execution. Return to monitor when instruction complete.
1	Like (2), but when finished with one instruction, immediately begin executing the next.
0	Like (1), but without display update.

Example: **STEP 3**

PRINTER

Turn printer on or off.

Syntax: **PRINTER ON/OFF**

Determines whether or not disassembly will be sent to a printer. Printing occurs after the simulator's execution of each instruction. If you have selected this option, a "P" will be present on the TVC Status Line.

If you don't **have** a printer, or if it is off-line, or if you have a non-standard interface, when you use this function, TVC will lock up, and you must reset to regain control.

Example: **PRINTER ON**

WINDOW

Set screen window.

Syntax: **WINDOW OPEN/CLOSE/MEM**

This command controls what is shown in the "window" area of the display (approximately the central third). There are three options: **CLOSE**, the default setting, displays the entire processor-Ram combination. **MEM** displays 16 selected memory locations. (See RC and LC functions). The programmer's registers (PC-A-X-Y-P-S) always remain on-screen. **OPEN** clears the area and leaves it blank.

Example: **WINDOW MEM**

ERASE

Erases display.

Syntax: **ERASE**

Clears entire display, but does not keep subsequent processes from writing to it. Step Mode 0 will help keep it clear.

Example: **ERASE**

RESTORE

Restores display.

Syntax: **RESTORE**

Undoes the work of the ERASE command by redrawing entire screen, according to the current window and register base settings. If issued while in Step Mode 0, the RESTORE command will redraw the screen and set the Step Mode to 1.

Example: **RESTORE**

LC/RC

Sets first address of right or left memory columns.

Syntax: **LC/RC <address>**

The effect of this command will not been seen unless the window is in MEM mode.

Example: **RC 900**

BSAVE

Save binary data to disk.

Syntax: **BSAVE <filename> [P3/P2]**

Master mode only. Saves binary data specified by the argument to the floppy disk in drive 1 under the name <filename>. You must give it a non-null name (i.e., something). The P3 (page three) argument saves $D0 bytes from $300 to $3CF. P2 saves from $200 to $2FF. If no argument is given, then BSAVE will save $400 bytes from $800-$BFF.

The standard DOS conditions must be met for this command to succeed. Namely, the drive door closed on an initialized, un-write protected DOS 3.3 disk with some room on it. **Do not attempt to defeat the write protection of the TVC disk.**

Examples: **BSAVE MAGNUMOPUS P3**
BSAVEMAGNUMOPUS.DATA

BLOAD

Load binary information from disk.

Syntax: **BLOAD <filename> [P3/P2]**

This command retrieves the programs and data stored by the BSAVE command. As with BSAVE, the default loading location is $800, and you are faced with the same set of DOS errors. And a special problem. BLOAD specifies the starting address a file is to be loaded at, not the length. If you load a file that is larger than TVC has room for, you will overwrite areas that TVC needs for itself.

After every BLOAD, TVC does a quick check to see if anything was overwritten. If it was, it attempts to boot to restore itself.

As a consequence of TVC's copy protection system, you will get I/O errors if you attempt a BLOAD from a standard DOS 3.3 disk from non master mode, or a BLOAD from the TVC disk in master mode.

If the window is currently in memory mode, BLOAD redisplays the window, even if it does not contain locations affected by the load. The next instruction line is also updated.

Examples: **BLOAD MAGNUMOPUS**
BLOAD HIRESTUFF P3

L

Disassemble Memory.

Syntax: **[<address>] L**

Disassembles 5 instructions beginning at <address>. If no address is specified, disassembly picks up where it left off previously.

Example: **800 L**
L

CALC

Turn on calculator.

Syntax: **CALC**

This command invokes a four function, three base, integer calculator. The four functions are +, -, *, and /. As with monitor commands, the operands and operator must be separated by spaces.

Your first keystroke has a special effect.

> A Control H, B, or D (Hex, Binary, Decimal) changes the calculator base and redisplays the number in that base.
>
> An **esc** exits back to the monitor (or to a simulator pause, depending on how you got here).
>
> Any other character clears the line and waits for your input.

To use the calculator for base conversion, enter the number you want converted, and use one of the base conversion keystrokes.

To convert $3CF into decimal: Set calculator base to hex with Ctrl-H. Enter **3CF return.** Your entry will be redisplayed with the cursor at the leftmost character. Enter ^D to see the number expressed in decimal. Or ^B for binary.

To multiply $3FF by $10, enter:

3FF * 10.

If the operation produces a value greater than 65,535 or a negative value less than -32,767 you will get a range error. Negative values are displayed in two's complement form.

Answers are displayed with the same routines that refresh the registers, and therefore include leading zeros and, with binary numbers, embedded spaces. You need not include leading zeros, and **must not** include spaces within numbers.

Example: **CALC**

EDIT

Edit memory.

Syntax: **EDIT <address>**

Entering EDIT mode displays the EDIT message, followed by the selected location and its contents. As with CALC mode, the first character entered has special significance.

return -- Display next location.

esc -- Exit edit mode to monitor.

left arrow -- Display previous location.

Any other character enters the standard input routine. When return is pressed, your entry is checked for validity in the current monitor base. If valid and within range, it replaces the value formerly at that address.

If that value is part of the instruction pointed at by the program

counter, the next instruction window will be updated.

See the discussion of the MASTER command for a description of what locations can be read and written to under various conditions.

Example: **EDIT 800**

LOAD REGISTER

Manually load register with selected value.

Syntax: **<register> <value>**

If one of 16 bit registers is specified,**<value>** can range from 0-65,535. Otherwise, loads greater than 255 produce range errors.

Example: **PC 300**

LOAD MEMORY

A shortcut to editing ram.

Syntax: **<address> <value>**

If **<address>** is a location that may be written to, **<value>** replaces the current contents of **<address>**. See the MASTER command for more information.

Example: **A00 FF**

BOOT

Boots disk.

Syntax: **BOOT**

Only way to exit TVC without cycling power. Boots the disk in slot six, drive one.

Example: **BOOT**

MASTER

Enters or exits master mode.

Syntax: **MASTER ON/OFF**

Master mode is for experienced users of TVC who desire more flexibility in debugging and executing programs. It is indicated by an "M" on the status line, and has the following effects:

1. Enables the GO command.
2. Enables the ZP command.
3. Enables the BSAVE command.
4. Enables the BLOAD command to read standard DOS 3.3 disks.
5. Allows reading and writing **all** memory locations, including those that would potentially alter or even crash TVC.
6. Changes interpretation of the BRK instruction (Opcode 00). In non-master mode, a BRK instruction causes the simulator to quit execution and return to the monitor. In step mode 0 and 1, BRK will change the step mode to 2. In master mode, BRK instructions are processed according to normal 6502 protocol.

Example: **MASTER**
MASTER OFF

ZP

Set zero page mode.

Syntax: **ZP SHARE/NOSHARE**

Master mode only. Controls where TVC will store zero page memory locations. If in SHARE mode, the "real" zero page used by Applesoft, DOS, the Monitor, and TVC itself is used. In noshare mode, you get 256 locations all to yourself. Defaults to NOSHARE. SHARE mode indicated by a Z on the monitor status line.

TVC uses zero page locations $06 through $09 and $FE-$FF. If you are displaying zero page addresses in ZP SHARE mode, locations changed by something other than the TVC monitor or simulator (such as Applesoft itself) will not change onscreen automatically; you must force the issue by executing RC/LC instructions to refresh the display.

Example: **ZP SHARE**

GO

Transfers program execution to the 6502.

Syntax: **GO**

Master mode only. If the next command is a JSR, execution of that subroutine is passed directly to the 6502.

Assuming the routine does no damage in the process of running, and there is no way TVC can protect itself in this situation, TVC will regain control when the 6502 executes the RTS at the end of the subroutine. This command is intended for use in quickly skipping over long known good routines, or to test in battle some code that seems to work correctly under TVC.

During 6502 execution, the message "6502 Mode" will appear on the error line of the monitor status area. When control is returned, the PC-A-X-Y-P registers, the disassembly window, and the next instruction line are updated.

More on GO in **How TVC Works.**

Example: **GO**

POP

Pop program counter from stack.

Syntax: **POP**

Simulates an RTS by loading the program counter with the stack's two topmost bytes and incrementing the stack pointer by two. The value placed in the PC as a result of this instruction will be meaningful only if the top of the stack contains the return address of the calling routine. If S contains $FE or greater, POP is ignored.

Useful as a way of backing out of slow, monotonous routines (such as a delay loop), or in figuring out how you came to be in a section of code.

Example: **POP**

Appendix D
6502 Simulator Reference

The 6502 simulator is the portion of The Visible Computer that runs 6502 machine language. It is indicated by the absence of the monitor prompt at the bottom of the display, and usually, a lot of on-screen activity. The simulator interactively executes the 151 defined instructions of the 6502 instruction set, by animating the microsteps necessary to perform each. Certain instructions (e.g., BRK) are executed differently depending on the setting of the MASTER flag. Undefined opcodes are trapped and refused.

THE MESSAGE WINDOW

If the simulator is active, the first line of the message window will display either "FETCH" (if the fetch cycle is in progress), or the mnemonic and addressing mode of the instruction under execution.

MICROSTEPS

The second line of the message window displays the "microstep" currently being executed. Microsteps are individual small tasks accomplished in sequence to complete a given instruction. The eight TVC microsteps are:

T: (transfer) Transfer a number from one register to another. The source register is unchanged.

READ Read into the data latch the contents of the address in AD.

WRITE Write the number in the data latch to memory location AD.

COMPUTE Do an arithmetic or logical operation.

COND FLAGS Condition the flags.

CALC ADDRS Use the X or Y registers to modify AD.

INC Increment a register.

DEC Decrement a register.

CONTROLLING THE SIMULATOR

The simulator is largely controlled by the monitor command STEP. The affect of each of the four step values is outlined here.

Step mode 3: The slowest, most instructive mode. The simulator pauses at each microstep. Pressing any key will cause execution to proceed to the next microstep. When the instruction is complete, the monitor is entered.

Step mode 2: Pauses do not occur automatically at microsteps. A pause may be forced by pressing any key **except** 1-9 and **esc**. The monitor is entered after the completion of a full instruction.

Step mode 1: Like mode (2) but instead of entering the monitor after completion of an instruction, the next instruction in memory is executed. **Esc** will force monitor entry after completion of the current instruction.

Step mode 0: Similar to step mode (1) **but without update of the display**. Only the disassembly and next instruction areas are kept current. As with (1), you can force monitor entry with **esc**. When you enter the monitor, the programmer's registers are updated to their proper values. Because this mode skips time consuming display routines, it gives the greatest execution speed, approximately .5 instructions per second.

SPEED CONTROL

The number keys control execution speed. **1** produces the fastest execution, **9** the slowest. They are ignored in step mode 0.

You can force the simulator to pause by typing any key except **esc**ape. Once in pause mode, pressing any key except **C** resumes execution. Pressing C puts TVC in calculator mode, the only monitor function available from within the simulator. Exiting

the calculator returns you to the pause state.

Appendix E
Error Messages

COMMAND The command interpreter cannot understand your instruction. Try again, and watch your syntax.

BASE TVC is unable to digest a numeric value you have fed it. Make sure you use values **valid in the selected base,** without embedded spaces.

RANGE You have entered a number too large for the situation. For example, trying to load the X register with $101.

NOT JSR A GO command has been issued without JSR as the next instruction.

NOT MASTER You have tried to execute a command only available under master mode, such as JSR or ZP.

DIV BY 0 The calculator was told to divide by zero.

BAD OPCODE The simulator was given one of the 104 undefined 6502 instructions.

DISK FULL You are trying to BSAVE a file to a disk that has no room for it. Use another disk.

NO FILE File not found. Check your spelling, and remember, no embedded spaces.

FILE LOCK You tried to BSAVE a file with the same name as a locked file.

W.PROTECT You attempted a BSAVE on a write protected disk. Note: do not attempt to defeat the write-protection of the TVC disk. Use an ordinary initialized DOS 3.3 diskette to save your files.

I/O Covers a multitude of sins related to disk operations, including: Drive doors left open, disks inserted upside down or not at all, uninitialized diskettes (or initialized by something other than DOS 3.3), and real problems like faulty or

glitched disks.

As a consequence of TVC's copy protection system, you will get I/O errors whenever you execute a BLOAD from a DOS 3.3 disk if you are not in master mode, or if you try to BLOAD from the TVC disk in master mode.

MISMATCH Type mismatch error. Happens when you try to BLOAD something besides a binary file, like an Applesoft program, or if you BSAVE a file under a name currently in use by a non-binary file.

ER XXXX-XX An internal error has occurred in TVC. This error is caused by either a bug in the program or something your activities in master mode have done to damage it. If you feel the first case is likely we'd like to know about it. Drop us a letter listing the **exact** error message and a thorough description of what you were doing when you got the error. You must reboot to recover from an internal error.

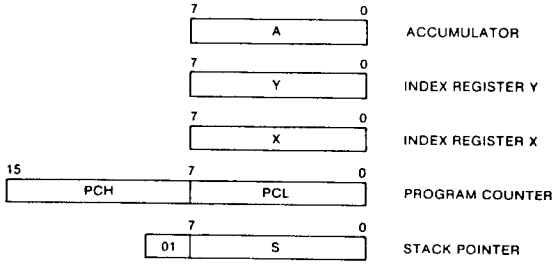
RESET AND CONTROL C

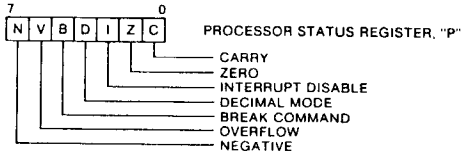
Never press Control C. This accomplishes nothing useful, and may cause TVC to function improperly afterwards.

Almost never press reset. Use it only as a last resort in situations such as when you have crashed the system by GOing a bugged subroutine, or trying to send disassembly to a non-existent printer. If you do press either of these keys, the display is redrawn and you are placed in the monitor.

Appendix F
6502 Reference Material

PROGRAMMING MODEL





THE FOLLOWING NOTATION APPLIES TO THIS SUMMARY:

A	Accumulator
X, Y	Index Registers
M	Memory
C	Borrow
P	Processor Status Register
S	Stack Pointer
✓	Change
—	No Change
+	Add
∧	Logical AND
-	Subtract
∀	Logical Exclusive Or
↓	Transfer From Stack
↓	Transfer To Stack
→	Transfer To
→	Transfer To
V	Logical OR
PC	Program Counter
PCH	Program Counter High
PCL	Program Counter Low
OPER	Operand
#	Immediate Addressing Mode

FIGURE 1. ASL-SHIFT LEFT ONE BIT OPERATION

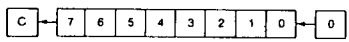


FIGURE 2. ROTATE ONE BIT LEFT (MEMORY OR ACCUMULATOR)

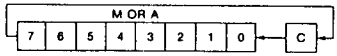
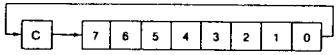


FIGURE 3.



Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C 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	6D	3	
		Absolute,X	ADC Oper,X	7D	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	2D	3	
		Absolute,X	AND Oper,X	3D	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	B0	2	------
BEQ Branch on result zero	Branch on Z=1	Relative	BEQ Oper	F0	2	------
BIT Test bits in memory with accumulator	A ∧ M, M_7 → N, M_6 → V	Zero Page Absolute	BIT* Oper BIT* Oper	24 2C	2 3	M_7√--M_6
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 ↓ P ↓	Implied	BRK*	00	1	---1--
BVC Branch on overflow clear	Branch on V=0	Relative	BVC Oper	50	2	------

* Bits 6 and 7 are transferred to the status register. If the result of A AND M is 0 then Z=1, otherwise Z=0.

* A BRK 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
BVS Branch on overflow set	Branch on V=1	Relative	BVS Oper	70	2	------
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
		Zero Page
		Zero Page, X
		Absolute
		Absolute,X
		Absolute,Y
		(Indirect,X)
		(Indirect),Y
			CMP #Oper
			CMP Oper
			CMP Oper,X
			CMP Oper
			CMP Oper,X
			CMP Oper,Y
			CMP (Oper,X)
			CMP (Oper),Y
				C9
				C5
				D5
				CD
				DD
				D9
				C1
				D1
					2
					2
					2
					3
					3
					3
					2
					2
						√√√---

CPX Compare memory and index X	X — M	Immediate Zero Page Absolute	CPX #Oper CPX Oper CPX Oper	E0 E4 EC	2 2 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	√√----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
EOR						
"Exclusive-Or" memory with accumulator	A V M → A	Immediate	EOR #Oper	49	2	√√ – –
		Zero Page	EOR Oper	45	2	
		Zero Page,X	EOR Oper,X	55	2	
		Absolute	EOR Oper	4D	3	
		Absolute,X	EOR Oper,X	5D	3	
		Absolute,Y	EOR Oper,Y	59	3	
		(Indirect,X)	EOR (Oper,X)	41	2	
		(Indirect),Y	EOR (Oper),Y	51	2	
INC						
Increment memory by one	M + 1 → M	Zero Page	INC Oper	E6	2	√√– – – –
		Zero Page,X	INC Oper,X	F6	2	
		Absolute	INC Oper	EE	3	
		Absolute,X	INC Oper,X	FE	3	
INX						
Increment index X by one	X + 1 → X	Implied	INX	E8	1	√√– – – –
INY						
Increment index Y by one	Y + 1 → Y	Implied	INY	C8	1	√√– – – –
JMP						
Jump to new location	(PC+1) → PCL	Absolute	JMP Oper	4C	3	– – – – – –
	(PC+2) → PCH	Indirect	JMP (Oper)	6C	3	
JSR						
Jump to new location saving return address	PC+2 ↓ (PC+1) → PCL (PC+2) → PCH	Absolute	JSR Oper	20	3	– – – – – –
LDA						
Load accumulator with memory	M → A	Immediate	LDA #Oper	A9	2	√√– – – –
		Zero Page	LDA Oper	A5	2	
		Zero Page,X	LDA Oper,X	B5	2	
		Absolute	LDA Oper	AD	3	
		Absolute,X	LDA Oper,X	BD	3	
		Absolute,Y	LDA Oper,Y	B9	3	
		(Indirect,X)	LDA (Oper,X)	A1	2	
		(Indirect),Y	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	
LDA						
Load accumulator with memory	M → A	Immediate	LDA #Oper	A9	2	√√– – – –
		Zero Page	LDA Oper	A5	2	
		Zero Page,X	LDA Oper,X	B5	2	
		Absolute	LDA Oper	AD	3	
		Absolute,X	LDA Oper,X	BD	3	
		Absolute,Y	LDA Oper,Y	B9	3	
		(Indirect,X)	LDA (Oper,X)	A1	2	
		(Indirect),Y	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	

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
LSR						
Shift right one bit (memory or accumulator)	(See Figure 1)	Accumulator Zero Page Zero Page,X Absolute Absolute,X	LSR A LSR Oper LSR Oper,X LSR Oper LSR Oper,X	4A 46 56 4E 5E	1 2 2 3 3	0√√----
NOP						
No operation	No Operation	Implied	NOP	EA	1	------
ORA						
"OR" memory with accumulator	A V M → A	Immediate Zero Page Zero Page,X Absolute Absolute,X Absolute,Y (Indirect,X) (Indirect),Y	ORA #Oper ORA Oper ORA Oper,X ORA Oper ORA Oper,X ORA Oper,Y ORA (Oper,X) ORA (Oper),Y	09 05 15 0D 1D 19 01 11	2 2 2 3 3 3 2 2	√√----
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	√√√----

Name Description	Operation	Addressing Mode	Assembly Language Form	HEX OP Code	No. Bytes	"P" Status Reg. N Z C I D V
RTI Return from interrupt	P↓ PC↓	Implied	RTI	40	1	From Stack
RTS Return from subroutine	PC↓, PC+1 → PC	Implied	RTS		1	------
SBC Subtract memory from accumulator with borrow	A - M - $\overline{C}$ → 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	------
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	√√----

This material reprinted from the **Apple II Reference Manual** through the courtesy of Apple Computer Inc.